Binary-Level Code Injection for Automated Tool Support on the ESP32 Platform

Benjamin Plach, Matthias Börsig[0000-0002-6060-6026], Maximilian Müller, Roland Gröll, Martin Dukek, and Ingmar Baumgart

FZI Research Center for Information Technology, Karlsruhe, Germany
publications@benjamin-plach.de,
{boersig,m.mueller,groell,dukek,baumgart}@fzi.de

Abstract. The analysis and testing of proprietary ESP32 firmware by independent security experts is often hampered by the lack of specialized tools that provide the necessary capabilities and ease of use to effectively support these tasks.
This paper presents a novel binary rewriting framework that addresses this challenge by allowing additional instructions to be inserted into ESP32 firmware without altering its original functionality. The framework leverages two already existing tools, Esptool and ESP32-Image-Parser, to extract firmware from ESP32 devices and convert it to ELF format, simplifying both the implementation of the framework and the development of subsequent tools.
In addition, an assembler has been developed to encode Xtensa assembly instructions without the need for linking the code afterward, facilitating the development of patch code. The framework includes a new patching methodology adapted from x86 patching tactics to the Xtensa architecture. These tactics have been implemented in a binary rewriting framework capable of inserting code at almost arbitrary locations without affecting the original firmware functionality.
A proof of concept tool that inserts fuzzing instrumentation was implemented to demonstrate the utility of the framework. This tool successfully integrates functional coverage information into ESP32 binaries. This framework represents a significant advancement in the tools available for firmware analysis and security testing of ESP32 devices.

Keywords: Static Binary Rewriting · Internet of Things · Embedded Systems · ESP32 · Xtensa · Microcontroller Security · Fuzzing

1 Introduction

The ESP32 platform has become increasingly prominent in the growing market for Internet of Things (IoT) devices due to its capabilities and cost-effectiveness. The proliferation of these devices has also led to an increased focus on their security. IoT devices are particularly vulnerable to security issues due to several

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025
L. Horn Iwaya et al. (Eds.): NordSec 2024, LNCS 15396, pp. 1–18, 2025.
https://doi.org/10.1007/978-3-031-79007-2_7

factors: The rush to bring products to market often results in inadequate security testing, there is a general lack of regular patching and the mechanisms for applying patches are often ineffective.

Independent security testing is important as it offers an unbiased evaluation of the security posture of devices. Manufacturers may overlook or underestimate vulnerabilities. Independent testing can circumvent these issues, providing a more thorough and objective assessment. Moreover, external testers, utilizing techniques like automated fuzzing, can probe systems in ways that internal teams might not consider, uncovering hidden vulnerabilities.

We propose to adopt a solution that involves the insertion of instrumentation into existing binaries via binary rewriting. This introduced approach automates the process of security testing by modifying the binary code of the firmware to include additional code that facilitates fuzzing and vulnerability detection. A crucial element of our method is utilizing binary rewriting while maintaining the original control flow. By preserving the original program flow, our technique ensures that the functionality and behavior of the firmware remain unchanged while still allowing the insertion of necessary instrumentation for effective fuzzing.

In this paper, we detail the method for binary-level code injection on the ESP32 platform. Our goal is to fill a gap in current security testing tools and provide a robust framework for improving the security of IoT devices built on the ESP32 platform.

Our contribution includes the development of a binary rewriting framework designed to modify and extend existing firmware binaries without disrupting their overall functionality. This includes the ability to insert patches into specific sections of the binary, primarily the `.flash.text` section. Additionally, we adapted patching tactics originally developed for the Complex Instruction Set Computer (CISC)-based x86-64 architecture to suit the Reduced Instruction Set Computer (RISC)-based Xtensa Instruction Set Architecture (ISA). We implemented a proof of concept to show the viability of these tactics. Lastly, our work includes assembler integration. We support a limited set of Xtensa ISA instructions and directives in the current assembler framework, with plans to expand this capability or integrate an external assembler to support a broader range of instructions.

2 Related Work

Duck et al. developed E9Patch, a static binary rewriter that capitalizes on the trampoline rewriting technique, which relies on the x86 long relative jump opcode [5]. E9Patch makes only minimal assumptions about the binary it alters, specifically avoiding dependencies on the binary's source language, compiler, debug information, or explicit control flow analysis. It does assume that patched instructions are neither accessed as data nor modified by the program itself, ruling out self-modifying code or overlapping instructions.

This approach has shaped our research direction, inspiring the method we adopted. Unlike E9Patch, which is tailored for x86-64 Linux binaries, our work focuses on adapting similar techniques to the Xtensa architecture of ESP32 devices, addressing specific challenges and leveraging unique opportunities in this context.

For the task of binary rewriting, various tools have been developed for different architectures and purposes. Performance optimization is the primary focus for tools like Lancet [12], Vulcan [1], and OM [15], while security hardening is targeted by others such as Zipr [8], RevARM [9], and CFI CaRE [11].

Binary rewriters are classified into dynamic and static types. Dynamic rewriters, like Dynamo [2], STRATA [14], and Pin [10], modify code at runtime, right before execution, which makes them unsuitable for embedding instrumentation into the firmware of ESP32 devices due to the need for pre-runtime modification.

Static binary rewriters, in turn, can be subdivided into categories based on their approach and functionality. Intermediate Representation (IR) rewriters such as mctoll [18] and revng [6] utilize LLVM as an intermediate language, enabling certain types of code manipulation that are beneficial for analysis and optimization. Meanwhile, reassemblable disassemblers like ddisasm [7], Retrowrite [4], and Uroboros [16] focus on the ability to disassemble and then reassemble binary code, offering a different set of capabilities. However, a common limitation across both categories is their inability to preserve the original structure of the binary. Maintaining the original structure is crucial for gaining useful insights from security testing.

Despite the rich ecosystem of binary rewriters detailed in comprehensive surveys by Wenzel et al. [17] and Schulte et al. [13], none of these existing approaches support the Xtensa architecture used in ESP32 devices. This highlights a gap that our work aims to address by adapting and extending binary rewriting techniques specifically for the Xtensa architecture.

3 Background

This section gives an overview about ESP23 firmware images and the code location problem that arises when trying to rewrite binaries as well as different categories of binary rewriting, i.e., static and dynamic rewriting.

3.1 ESP32

The ESP32 is a versatile, low-power and cost-effective microcontroller family from Espressif Systems, aimed at IoT applications. As an advancement over the ESP8266, it offers enhanced functionality with various options such as single or dual Xtensa LX6 cores, Xtensa LX7 cores, or a RISC-V core, along with various memory configurations. The ESP32 also features integrated Wi-Fi and Bluetooth, alongside a set of peripherals such as touch sensors, SD card interfaces, and Ethernet.

Firmware Images An overview of the ESP32 firmware image can be found in Fig. 1. It contains three main components: the bootloader, the partition table,

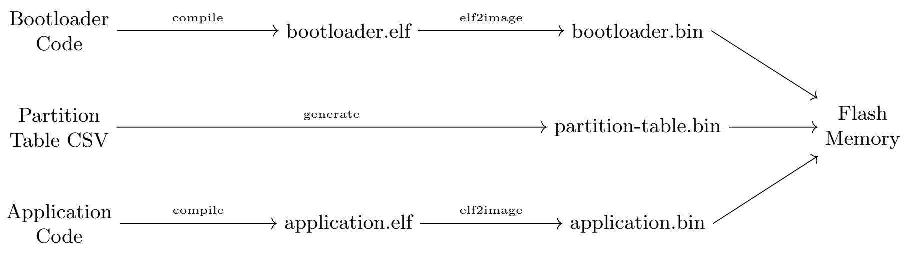


Fig. 1. The build and flash process

and at least one app partition containing the application code. The bootloader and application code are compiled into Executable and Linkable Format (ELF) binaries and are then converted into a binary format suitable for the ESP32.

These binaries are flashed together to the microcontroller. Although they are not combined into a single image, the bootloader, partition table, and application image are collectively referred to as the firmware image in this paper. Modifying or swapping any of these components typically results in a failure of the firmware to function correctly.

Bootloader The bootloader runs first when the ESP32 is powered up or reset. It initializes the hardware, sets the clock, configures the memory, and verifies the integrity of the application code before passing control to the application.

Partition Table The partition table defines the layout of the ESP32's flash memory, including the location and size of the bootloader, the application, and other partitions.

Application Image The application image is the main firmware that contains the code for the functionality of the ESP32. After the bootloader initializes the system, it transfers control to the application image, which then carries out the device's intended operations.

Xtensa ISA The Xtensa ISA is a customizable and extensible RISC-based ISA designed to be configurable via different options. These allow designers to tailor the instruction set to specific performance, power, and area requirements by extending it with custom instructions and registers.

While the Xtensa ISA follows most of the typical RISC conventions, such as load/store architecture or single-cycle instructions, it deviates from on in particular. When the "Code Density Option" is activated, which is almost always the case, additional 16-bit versions of the regular 24-bit instructions are introduced. This option enables more efficient memory usage, an important advantage in embedded systems where memory resources are often limited. In an average ESP32 binary around half of the instructions are 16-bit instructions.

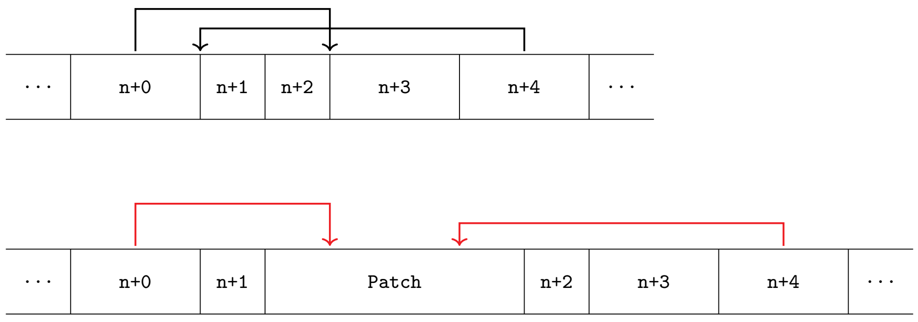


Fig. 2. The code location problem

The "Windowed Register Option" introduces additional general purpose registers, bringing the total number from 16 to 64. To avoid having to introduce new instructions with larger register encodings (from 4 to 6 bit per register used in the instruction), the number of visible registers stays at 16. During function calls this window of visible registers can be shifted by a multiple of 8. This option reduces the amount of registers that have to be saved on the stack before a function call.

3.2 The Code Location Problem

The code location problem arises when modifying the binary code of a program. This issue is prevalent in any system where instructions use relative addressing, such as jump and branch instructions that calculate their targets based on their position in memory. When new instructions are inserted or existing ones are altered, these changes can shift the positions of subsequent instructions (as seen in Fig. 2), potentially causing incorrect jumps, crashes, or unpredictable behavior.

3.3 Binary Rewriting

Binary rewriting can be broadly categorized into static and dynamic methods, each with distinct approaches and trade-offs.

Static rewriting involves modifying the binary code without executing it, resulting in a new, modified binary file. This method allows for comprehensive analysis and optimization prior to execution. However, it has the disadvantage that jump targets within the code must be recalculated without runtime information or kept at their current positions, making it difficult to make changes to the code. It also struggles with self-modifying code.

Dynamic rewriting modifies the binary code during execution, enabling adjustments based on real-time behavior and conditions. While it is better suited to handle dynamic and self-modifying code, it introduces runtime overhead, which can negatively impact performance.

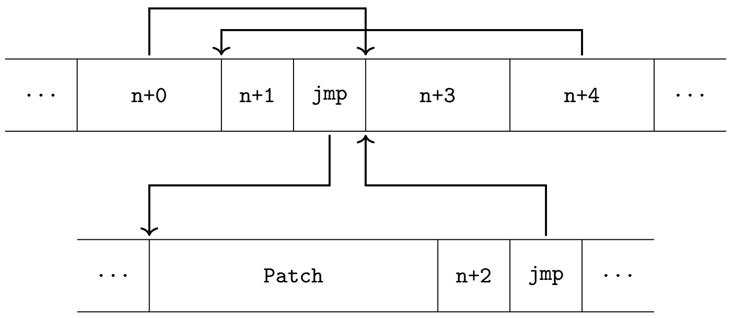


Fig. 3. Trampoline rewriters offer a solution to the code location problem.

For IoT devices, static rewriting is generally preferred because dynamic rewriting often relies on virtualization. Although there is a QEMU implementation for the Xtensa architecture, it does not support network interfaces, which is an important feature for most IoT applications.

Static rewriters can either convert the binary to an IR, like assembly, make changes at that level of abstraction, and then reassemble the binary. While this is a small overhead, the original control flow is lost, and new bugs may be introduced.

Static trampoline rewriters place patches in unused areas of the binary and redirect control flow to those patches using jump instructions, as shown in Fig. 3, this allows the original order of control flow to be preserved.

4 Design

Our design consists of several steps: extracting of the binary, rewriting the binary by applying different patching tactics, and reflashing the modified binary.

4.1 Binary Recovery

The binary recovery starts with extracting a complete flash dump of the ESP32 device. Following this, we recover the partition table and use its information to identify the application binary. A crucial transformation in this process is converting the recovered binary image into an ELF file format, which simplifies subsequent analysis and modification tasks. The general idea of this extraction and reflashing process is shown in Fig. 4.

4.2 Rewriter

The rewriter consists of several patching tactics, solving individual patch cases, and a broader binary-wide strategy. The tactics are presented in the order of their attempted application, e.g., if the jump tactic fails to apply the patch, the punned jump tactic is tried next.

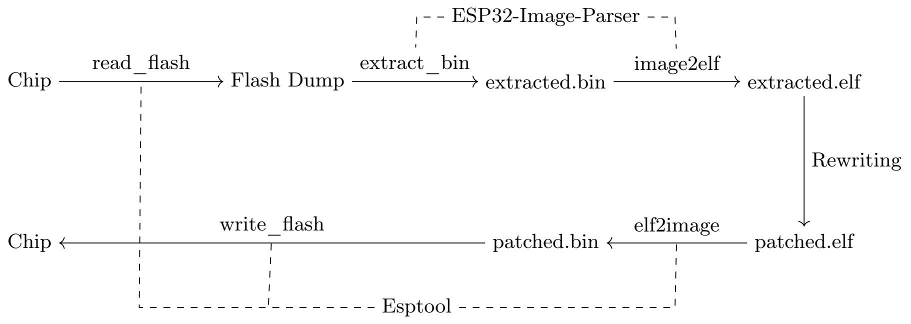
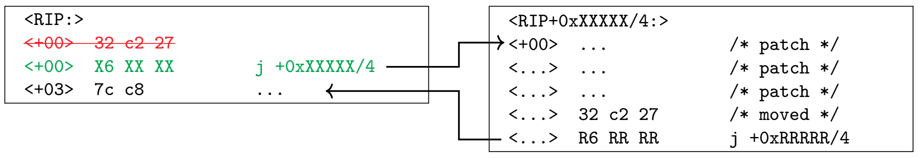


Fig. 4. Binary recovery, rewriting, and re-flashing process

Patching Tactics The rewriter component is responsible for modifying the recovered ELF binary to insert the necessary instrumentation. We use various patching tactics to accomplish this without disrupting the original program flow.

Jump Tactic The jump tactic involves redirecting the control flow from the original code to the newly inserted instrumentation and then back to the original code. This is accomplished by inserting jump instructions instead of the original instruction that is moved to the trampoline code. Fig. 5 illustrates the application of the jump tactic to the Xtensa architecture, using the following syntax: The original instruction (red) is removed and replaced by the jump instruction (green), while the `X` represents an arbitrary selectable value. However, the opcode of the jump instruction is six bits, leaving 18 bits for the relative offset to address the target location. These 18 bits are displayed as five `X`s, each representing a half-byte in hexadecimal encoding, but is followed by a `/4` (right shift by 2 bits) to indicate that the two Most Significant Bits (MSBs) are cut off because they are part of the opcode. This jump instruction now points to the beginning of the trampoline (right side) where the patch is inserted, including the original instruction at the end. The last instruction in the trampoline points to the first instruction after the jump.

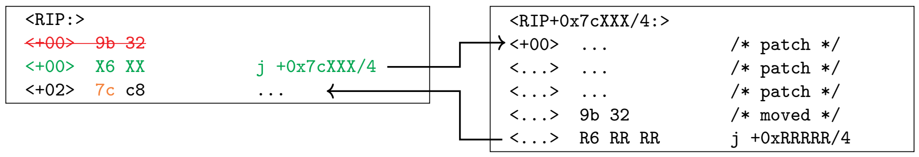


```
<RIP:>
~~<+00>  32 c2 27~~
<+00>  X6 XX XX        j +0xXXXXX/4
<+03>  7c c8           ...
```

```
<RIP+0xXXXXX/4:>
<+00>  ...             /* patch */
<...>  ...             /* patch */
<...>  ...             /* patch */
<...>  32 c2 27        /* moved */
<...>  R6 RR RR        j +0xRRRRR/4
```

Fig. 5. Applying the jump tactic on Xtensa

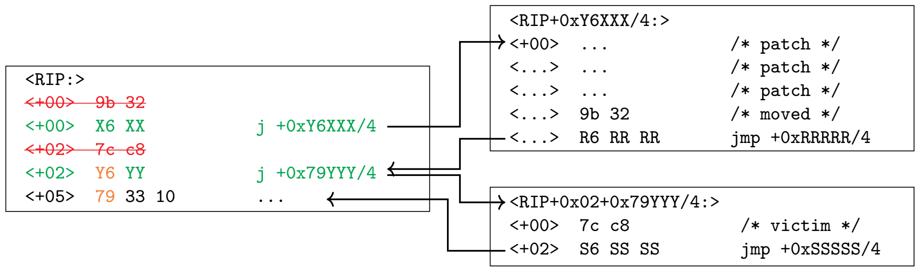
Punned Jump Tactic The punned jump tactic is a variation of the jump tactic used when the target instruction that would be replaced with the jump instruction is a short 16 bit instruction and not a 24 bit instruction. In such cases, the initial byte of the subsequent instruction can be incorporated into the current instruction, a technique known as instruction punning [3]. Fig. 6 demonstrates the punned jump tactic. In addition to the explanation above, we now have punned bytes of the following instruction (orange), which cannot be changed and limit the range of our jump command. Depending on the program, it may be harder to find free space for the trampoline. This limitation is deliberately accepted in order to open up new possibilities, allowing us to insert the jump in tight spaces and use the patch in situations that would otherwise be impossible.



```
<RIP:>                                   <RIP+0x7cXXX/4:>
~~<+00>  9b 32~~                          <+00>  ...          /* patch */
<+00>  X6 XX          j +0x7cXXX/4       <...>  ...          /* patch */
<+02>  7c c8          ...                <...>  ...          /* patch */
                                         <...>  9b 32        /* moved */
                                         <...>  R6 RR RR     j +0xRRRRR/4
```

Fig. 6. Applying the punned jump tactic on Xtensa

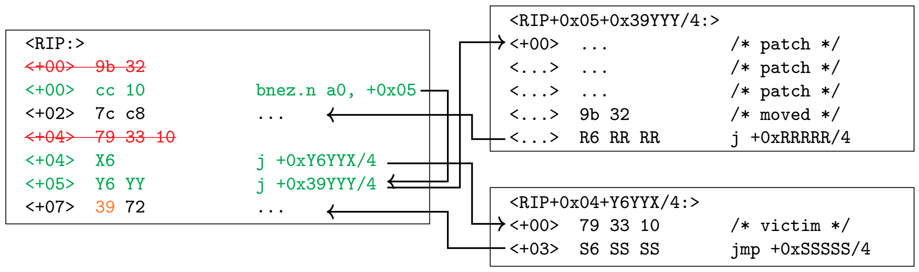
Successor Eviction Tactic If the previous tactic is unsuccessful, the subsequent eviction tactic, which also moves the next instruction to a different code area (using any of the above tactics), can be used. Moving both the original instruction and its successor gives us additional options for finding unused code locations. The displaced instructions are relocated to a new address within the binary, and the control flow is adjusted to ensure the program continues to execute correctly. Fig. 7 demonstrates this tactic. The main difference from the last patch tactic is that we now have two replacements, giving us options in cases where the instruction punning tactic fails to find a suitable area for the trampoline code.



```
                                         <RIP+0xY6XXX/4:>
                                         <+00>  ...          /* patch */
                                         <...>  ...          /* patch */
<RIP:>                                   <...>  ...          /* patch */
~~<+00>  9b 32~~                          <...>  9b 32        /* moved */
<+00>  X6 XX          j +0xY6XXX/4       <...>  R6 RR RR     jmp +0xRRRRR/4
~~<+02>  7c c8~~
<+02>  Y6 YY          j +0x79YYY/4       <RIP+0x02+0x79YYY/4:>
<+05>  79 33 10       ...                <+00>  7c c8         /* victim */
                                         <+02>  S6 SS SS      jmp +0xSSSSS/4
```

Fig. 7. Applying the successor tactic on Xtensa

Neighbor Eviction Tactic The neighbor eviction tactic is another possible option if the successor eviction tactic fails and is similar to it, but it evicts an instruction after the patch point. This approach, shown in Fig. 8, provides even more flexibility in finding unused code locations. As the Xtensa ISA does not provide a short relative jump, this tactic exploits the `bnez.n` branch instruction, a 16-bit instruction that performs a relative jump when a register is not zero. The `a0` register holds the return address, and thus should never be zero, resulting in a guaranteed branch.



```
<RIP:>
~~<+00>  9b 32~~
<+00>  cc 10            bnez.n a0, +0x05
<+02>  7c c8            ...
~~<+04>  79 33 10~~
<+04>  X6               j +0xY6YYX/4
<+05>  Y6 YY            j +0x39YYY/4
<+07>  39 72            ...
```

```
<RIP+0x05+0x39YYY/4:>
<+00>  ...              /* patch */
<...>  ...              /* patch */
<...>  ...              /* patch */
<...>  9b 32            /* moved */
<...>  R6 RR RR         j +0xRRRRR/4
```

```
<RIP+0x04+Y6YYX/4:>
<+00>  79 33 10         /* victim */
<+03>  S6 SS SS         jmp +0xSSSSS/4
```

Fig. 8. Applying the neighbor eviction tactic on Xtensa

Patching Strategy For every patch, the tactics are tried sequentially, until one can successfully apply the patch. If the last one fails, the patch can not be applied.

All patches are applied in reverse order, meaning that patching starts from high addresses and moves to lower ones. This behavior avoids "locking in" the following bytes that might need to be changed too. To better understand this, imagine the second instruction in the punned jump tactic example in Fig. 6 needed to be moved for a patch too. If the first instruction was patched first (normal order), the first byte of the second instruction would be "locked in" and a patch could not be applied. If this second instruction is patched first (reverse order), the instruction punning of the first instruction might still succeed with the new byte `Y6` as punned byte.

5 Implementation

The following shows the implementation of design discussed in the previous section. Our implementation is specifically designed to work on the ESP32-WROOM-32 model, which is a widely used version of the ESP32.

5.1 Binary Recovery

First, a complete flash dump of the target ESP32 device is extracted using Esptool[1]. Afterward, the ESP32-Image-Parser[2] is used to locate and extract the application image from the flash dump, and to transform it into the ELF format. However, this tool is outdated and requires several bug fixes. A fixed version with extended functionality can be found on GitHub[3].

5.2 Rewriter

The rewriter tool is our main contribution and is used for modifying the recovered binary to insert the necessary instrumentation for tasks such as fuzzing. Fig. 9 shows the relationship to the other components.

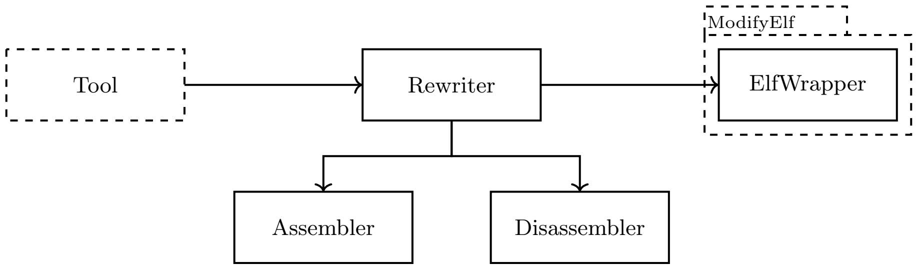


Fig. 9. The structure of our approach

The rewriter is structured to support future tool development. We implemented an adapter for the Radare2 disassembler and a new assembler. The ModifyElf library allows manipulating ELF binaries. It encapsulates the complexities of the ELF format and provides both, a low-level and a high-level interface. The low-level interface is provided by the `ElfRaw` class, which allows detailed and precise modifications, while the `ElfWrapper` class provides a more abstract interface for ease of use.

Patching Tactics Each patching tactic takes the patch location and patch code as input and returns whether the patching attempt was successful or not. An optional parameter allows the moved statement to be executed before or after the patch code, with the default being execution before the patch code.

[1] `https://github.com/espressif/esptool/`
[2] `https://github.com/tenable/esp32_image_parser`
[3] `https://github.com/benjamin-plach/esp32-binary-rewriting-paper`

Patching Strategy The patching strategy function takes a list of patches and attempts to apply them in reverse order to their target location. For each patch, the function sequentially applies the available patching tactics. Currently, it first attempts the jump tactic, followed by the punned jump tactic.

The method provides feedback on the success of each patch attempt, and summarizes the coverage information at the end.

Assembler The assembler produces code that requires no further linking. It takes a start address along with the instruction stream as input, allowing the assembler to correctly encode instructions with relative offsets such as jumps or relative load instructions.

The assembler supports the basic features of assembly language coding: the encoding of several Xtensa assembler instructions from the Xtensa ISA Summary[4], assembler directives such as `.align 4` for 4-byte alignment, labels for code locations, and comments.

5.3 Flashing the Rewritten Binary back onto the Device

Once the rewriting is complete, the binary can be converted from the ELF format back into the ESP application image format and flashed back on the device.

The address specified in the rewrite command must be the same address from which the binary was recovered. Otherwise, the bootloader will not find it. This address can be looked up in the recovered partition table.

6 Proof of Concept

The binary rewriter has several applications, such as inserting arbitrary code or applying third-party security patches without changing the original program's control flow. To demonstrate its potential, we have developed a proof-of-concept tool that focuses on fuzzing instrumentation. This tool is designed to showcase the potential of the binary rewriter. Consequently, it focuses on collecting coverage information for function calls only, without tracking branches or loops.

6.1 Designing an Example Tool

Three options were considered for implementing the counters needed to collect fuzzing coverage information:

– **Flash Memory:** Using an unused area of flash memory to store counter variables can offer performance benefits. However, this approach introduces risks like losing the information stored in the memory to a power loss or device crash.

[4] `https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/silicon-solutions/compute-ip/isa-summary.pdf`

- **Non-Volatile Storage (NVS) Partition:** Adding an NVS partition to the firmware to store counter information can be an efficient solution. However, it requires modifying the partition table and implementing NVS access in assembly language.
- **Monitoring Messages:** Sending unique function identifiers to a connected device through monitoring messages allows for function call counting. On the ESP32, data sent to `stdout` and `stderr`, e.g., via `printf`, is forwarded to a monitoring device. This option is the easiest to implement but can not be used to get coverage information for all functions, as inserting a counter at the start of the chosen function, e.g., `printf`, and any function it calls would result in an infinite loop.

For this proof of concept, we have chosen to implement monitoring messages using the `printf` function, which is a natural choice given the task. While fuzzing standard library functions like `printf` is important for comprehensive software security, the inability to do so here is a negligible disadvantage since our primary focus is on custom application code.

6.2 Implementing the Example Tool

Strings on ESP32 On the ESP32, strings are stored in the `.flash.rodata` section. The Xtensa ISA's 24-bit instruction size does not allow to directly encode 32-bit absolute addresses and the `l32r` instruction used for loading 32-bit values has a limited range. Therefore, pointers to strings are placed at the beginning of the `.flash.text` section and loaded into registers using the `l32r` instruction before calling `printf`.

Listing 1.1. Strings on ESP32

```
<.flash.rodata:>
0x3f4041a8:    48 65 6c 6c 6f   ; Hello
               20 77 6f 72 6c   ;  Worl
               64 21 00         ; d!\0

<.flash.text:>
0x400d0618:    a8 41 40 3f      ; pointer to 0x3f4041a8

0x400d500f:    a1 82 ed         ; l32r a10, -0x049f7
0x400d5012:    e5 77 05         ; call8 <printf>
```

Patch Code Initial attempts to use custom strings in the patch code failed, likely due to memory access restrictions or alignment issues that prevent the hardware from reading bytes directly from the `.flash.text` section. However, leveraging existing strings within the binary – such as those used by system functions – proved to be an effective alternative for instrumentation purposes.

It is important to note that while the ability to insert custom strings into the binary might be desirable in certain scenarios, it is not a critical requirement for many forms of security testing, such as fuzzing or coverage-based instrumentation. The primary objective of our rewriter is to inject observational instructions without disturbing the control flow of the firmware, and this goal is achieved regardless of the source of the strings. Therefore, the use of pre-existing strings offers a practical solution without affecting the utility or effectiveness of the framework.

Consequently, existing strings in the binary were used, with several criteria:

– **Availability:** The string must be present in all ESP32 binaries. Strings in FreeRTOS, a small operating system commonly used in ESP32 applications, that are compiled into ESP32 code fulfill this criterion.
– **Position:** The pointer to the string must be close enough for the `l32r` command to load it.
– **Structure:** The string must accept a 32-bit integer as its last parameter. This ensures that data, containing the current address of the function call, can be output directly without requiring additional conversions.

The string “`W (%lu) %s: Flash clock frequency round down to %d`” has been selected and cut. Its pointer and the address of the `printf` function must be present when the tool is initialized.

The core function `add_fuzzing_counter` assembles the patch code, loads the string pointer and truncates the string, loads the counter address and calls `printf`.

Listing 1.2. Defining the patch

```
def add_fuzzing_counter(self, addr:int):
    fuzzing_counter_patch = [
        "             l32r a10, " + hex(self.
            __addr_string_pointer),
        "             addi a10, a10, 52", # trim start
            of string
        "             j jmplabel", # jump over data
        "             .align 4",
        "addrlabel:   .uint32 " + hex(addr),
        "jmplabel:    l32r a11, addrlabel", # always -4
        "             call8 " + hex(self.
            __addr_printf_function)
    ]

    self.rewriter.add_patch(addr, fuzzing_counter_patch
        , moved_after_patch=True)
```

The registers `a10` and `a11` can be utilized without the risk of overriding meaningful data, because the register window was just shifted by 8 during the function call.

Patches are applied to the `entry` instruction, which marks the starting point of each function. This ensures that every counter is only triggered once per function call, as the control flow will never be redirected to the `entry` instruction within one function call.

Monitoring A monitoring script is implemented to collect `printf` output on the monitoring device, filtering data from the `stdout` stream and counting addresses sent by the patch code. After a specified time, the counting is stopped and the results are displayed.

6.3 Utilizing the Example Tool

The example tool can now be utilized to insert fuzzing instrumentation into an existing binary. After reflashing it back onto the device, the monitoring script is used to collect the results.

First, the tool must be initialized by providing the path to the extracted binary. Once initialized, the fuzzing counters can be added. After loading the binary, fuzzing counters can be added. Currently, the user must identify the addresses where the counters should be placed himself, e.g., by using a disassembler. These addresses can then be added using the `add_fuzzing_counter` method. In addition, the user must inform the framework where additional code can be safely inserted without overwriting existing functionality using the `add_free_space` method. This step is critical to maintaining the integrity and proper execution of the original program.

Once these steps have been completed, the patches can be applied. This step handles the actual insertion of the patches into the binary, ensuring that the new instructions are correctly placed and aligned with the existing code structure, and that any jumps are correctly targeted. Finally, the rewritten ELF binary needs to be saved to a file, that can be flashed back onto the device.

Listing 1.3. Using the example tool

```
inserter = Fuzzing_Instrumentation_Inserter('extracted.
    elf')

inserter.add_fuzzing_counter(0x400e248c)
inserter.add_fuzzing_counter(0x400d4fc0)
inserter.add_fuzzing_counter(0x400d4fdc)

inserter.add_free_space(0x400e23a8, 0x400e2489)

inserter.apply_patches()
util.save_file("patched.elf", inserter.get_elf_bytes())
```

Once the binary is rewritten and flashed back to the device, the monitoring script can be run to collect the coverage information. The script tracks the `printf` calls made by the patch code. After the run is completed the results are displayed.

Listing 1.4. Running the monitoring script

```
 > python3 monitoring.py
[COUNTER] 0x400d4fdc
[COUNTER] 0x400d4fc0
  [...]
[COUNTER] 0x400e248c
[COUNTER] 0x400e248c

[CONTROL] Terminating Run!

[RESULTS] 0x400d4fdc: 1
[RESULTS] 0x400d4fc0: 5
[RESULTS] 0x400e248c: 15
```

7 Limitations

Several limitations were encountered during this research. A primary constraint is the limited space available on the ESP32 device for patches. While small patches can be accommodated in padding areas, larger modifications may require extending existing code sections within the binary or modifying the bootloader to load additional code at boot time.

One set of instructions that can currently not be moved are instruction with relative offsets encoded into them, but the relocation of these instructions would require a recalculation of their offsets. Avoiding this is a core design philosophy of trampoline rewriters.

The current implementation only supports a limited set of assembler instructions, restricting the complexity of patches that can be applied. Additionally, the current patching strategies are limited to the Jump and Punned Jump tactics. Implementing the Successor Eviction and Neighbor Eviction tactics, which both require a more in-depth disassembler integration, will increase the rate of successfully applied patches.

8 Future Work

Future work could focus on addressing the identified limitations and enhancing the capabilities of the binary rewriter. One potential area of improvement is the automatic extension of binary sections, particularly the `.flash.text section`, to create additional space for patching without affecting other parts of the binary. Additionally, expanding assembler support to include the full range of Xtensa ISA instructions, or integrating an external assembler, would provide greater flexibility in applying more complex patches.

The patching tactics Successor Eviction and Neighbor Eviction were not necessary for our proof of concept tool, and therefore not implemented. More sophisticated tools would greatly benefit from implementing them. Besides these two tactics, exploring new patching tactics that leverage specific features of the Xtensa architecture could also increase the efficiency of the rewriter.

The reverse-order patching strategy, while effec tive, may not be optimal in all scenarios. Improving patching strategies, by experimenting with heuristic-based or randomized approaches, could further enhance the success rate of patch applications.

The fuzzing instrumentation inserter was chosen as a proof of concept because it is a promising use case to utilize the binary rewriting framework. As showcased, the tool shows great potential for third-party security testing, but needs to be developed further before it can be applied in real-world scenarios. This could include automated detection of the fuzzing instrumentation locations, and the addition of counters for loops and branches.

9 Conclusion

This paper addresses the gap in tool support for independent security experts to analyze and test proprietary ESP32 firmware. We propose a binary rewriting framework that enables the insertion of additional instructions into existing ESP32 firmware without altering its original functionality and control flow.

The framework simplifies the process of firmware analysis and modification by converting the extracted firmware into a more manageable format, allowing for precise changes while maintaining the integrity of the original code. It introduces novel patching methodologies tailored to the Xtensa architecture, adapting established techniques to meet the specific needs of ESP32 devices. The effectiveness of the framework was demonstrated by a proof of concept fuzzer that successfully added coverage information to ESP32 binaries. This approach involves inserting a counter that records the number of times a particular section of code is executed. By using this feedback, the fuzzer can explore different execution paths more efficiently, increasing the likelihood of finding bugs and vulnerabilities. This demonstrates the potential for further development of the framework to improve the security of ESP32 firmware.

Future work should focus on enhancing the versatility of the framework by implementing additional patching tactics and developing new ones, especially to address more complex patching scenarios. Continued refinement of this framework will expand its capabilities and further support the security analysis of ESP32 devices.

References

1. Amitabh Srivastava, Andrew Edwards, H.V.: Vulcan: Binary transformation in a distributed environment (2001), https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2001-50.pdf
2. Bala, V., Duesterwald, E., Banerjia, S.: Transparent dynamic optimization: The design and implementation of dynamo (1999), https://homes.cs.washington.edu/~bodik/ucb/cs703-2002/papers/dynamo-full.pdf
3. Chamith, B., Svensson, B.J., Dalessandro, L., Newton, R.R.: Instruction punning: lightweight instrumentation for x86-64. SIGPLAN Not. **52**(6), 320–332 (jun 2017), https://doi.org/10.1145/3140587.3062344
4. Dinesh, S., Burow, N., Xu, D., Payer, M.: Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 1497–1511. IEEE (2020), https://doi.org/10.1109/SP40000.2020.00009
5. Duck, G.J., Gao, X., Roychoudhury, A.: Binary rewriting without control flow recovery. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 151–163. ACM (2020), https://doi.org/10.1145/3385412.3385972
6. Federico, A.D., Payer, M., Agosta, G.: rev.ng: a unified binary analysis framework to recover cfgs and function boundaries. In: Wu, P., Hack, S. (eds.) Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017. pp. 131–141. ACM (2017), http://dl.acm.org/citation.cfm?id=3033028
7. Flores-Montoya, A., Schulte, E.M.: Datalog disassembly. In: Capkun, S., Roesner, F. (eds.) 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020. pp. 1075–1092. USENIX Association (2020), https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya
8. Hawkins, W.H., Hiser, J.D., Co, M., Nguyen-Tuong, A., Davidson, J.W.: Zipr: Efficient static binary rewriting for security. In: 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017. pp. 559–566. IEEE Computer Society (2017), https://doi.org/10.1109/DSN.2017.27
9. Kim, T., Kim, C.H., Choi, H., Kwon, Y., Saltaformaggio, B., Zhang, X., Xu, D.: Revarm: A platform-agnostic ARM binary rewriter for security applications. In: Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017. pp. 412–424. ACM (2017), https://doi.org/10.1145/3134600.3134627
10. Luk, C., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005. pp. 190–200. ACM (2005), https://doi.org/10.1145/1065010.1065034
11. Nyman, T., Ekberg, J., Davi, L., Asokan, N.: CFI care: Hardware-supported call and return enforcement for commercial microcontrollers. CoRR **abs/1706.05715** (2017), http://arxiv.org/abs/1706.05715
12. Put, L.V., Sutter, B.D., Madou, M., Bus, B.D., Chanet, D., Smits, K., Bosschere, K.D.: LANCET: a nifty code editing tool. In: Ernst, M.D., Jensen, T.P. (eds.) Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis

For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005. pp. 75–81. ACM (2005), https://doi.org/10.1145/1108792.1108812

13. Schulte, E., Brown, M.D., Folts, V.: A broad comparative evaluation of x86-64 binary rewriters. In: CSET 2022: Cyber Security Experimentation and Test Workshop, Virtual Event, 8 August 2022. pp. 129–144. ACM (2022), https://doi.org/10.1145/3546096.3546112
14. Scott, K., Kumar, N., Velusamy, S., Childers, B.R., Davidson, J.W., Soffa, M.L.: Retargetable and reconfigurable software dynamic translation. In: Johnson, R., Conte, T., Hwu, W.W. (eds.) 1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003), 23-26 March 2003, San Francisco, CA, USA. pp. 36–47. IEEE Computer Society (2003), https://doi.org/10.1109/CGO.2003.1191531
15. Wall, D.W., Srivastava., A.: A practical system for intermodule code optimization at link-time (1992), https://web.stanford.edu/class/cs343/resources/om.pdf
16. Wang, S., Wang, P., Wu, D.: Reassembleable disassembling. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 627–642. USENIX Association (2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-shuai
17. Wenzl, M., Merzdovnik, G., Ullrich, J., Weippl, E.R.: From hack to elaborate technique - A survey on binary rewriting. ACM Comput. Surv. **52**(3), 49:1–49:37 (2019), https://doi.org/10.1145/3316415
18. Yadavalli, S.B., Smith, A.: Raising binaries to LLVM IR with MCTOLL (WIP paper). In: Chen, J., Shrivastava, A. (eds.) Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2019, Phoenix, AZ, USA, June 23-23, 2019. pp. 213–218. ACM (2019), https://doi.org/10.1145/3316482.3326354