HAW HAMBURG

Project Report

Lars Pfau

Integrating a RISC-V Secure Firmware into RIOT

Supervision: Prof. Dr. Thomas C. Schmidt
Submitted: June 21st, 2024

Faculty of Engineering and Computer Science
Department Computer Science

Contents

Abstract This work investigates the possibility of reducing the code size of trusted execution environments in constrained IoT devices. The goal of this work is to integrate secure firmware into the real-time operating system RIOT while minimizing the duplication of libraries such as for cryptography. First, an analysis is performed that identifies numerous problems related to code sharing between execution contexts. Second, some solutions to these problems are investigated and an approach using the PSA Crypto API is selected. Finally, a separate secure firmware and PSA Crypto driver are developed and integrated into RIOT. The results show significant code size advantages from deduplicating crypto algorithms using the PSA Crypto API.

Keywords: IoT Security, Trusted Execution Environments, RISC-V

1 Introduction

IoT security is a research area concerned with the security of constrained devices. The development of such IoT nodes is driven by cost and power consumption, resulting in hardware with limited computing and memory resources. RFC 7228 [17] defines highly constrained IoT devices as those with less than 10 KiB of SRAM and less than 100 KiB of flash memory. Until recently, such constrained IoT devices consisted of microcontrollers that run bare-metal software and did not implement memory isolation. This meant that all memory and peripherals could be accessed by all software. As a result, simple buffer overflows could lead to information disclosure and remote code execution. With the introduction of Arm TrustZone-M [12] and RISC-V PMP [34], microcontrollers based on the ARMv8-M and RISC-V architecture optionally support hardware efficient memory isolation mechanisms. This has led to research interest to implement trusted execution environments on IoT devices.

The idea of trusted execution environments is to separate an application into a normal execution environment and one or more secure execution environments. The traditional bare-metal application runs in the normal execution environment with limited memory and peripheral access, while secure firmware and enclaves run in a secure execution environment. The secure environment guarantees properties such as integrity, authenticity, and confidentiality of code and data which can be used to implement features like secure boot, remote attestation, key stores, sealed storage, and more, increasing the security of low-end Internet-connected devices.

However, the implementation of trusted execution environments on constrained devices faces code-size challenges. TEE solutions such as MultiZone [19] work by compiling and linking the applications and the secure monitor independently. This can result in code duplication of common libraries between execution environments, such as for cryptography, potentially wasting scarce flash memory space.

This report analyzes the potential for reducing redundant code between secure and normal execution environments. The goal of this work is to integrate a secure firmware based on the method described in [27] into the real-time operating system RIOT [14].

2 Related Work

This project report is a continuation of the work in [27]. In the previous work, an experiment was performed that showed the feasibility of implementing trusted execution environments on RISC-V microcontrollers with the RIOT operating system. In the experiment, the RIOT RISC-V port was modified to be able to run in user-mode. A minimal machine-mode secure firmware was developed to handle system startup, interrupt handling, and access to privileged instructions. This experimental secure firmware was integrated directly into the repository and build system of RIOT. A modified linker script was used to group all protected code and data sections together to isolate them from unprivileged access using RISC-V PMP.

While this approach was very simple and effective for the purposes of the experiment, it is important to understand that it only worked because the machine-mode firmware was written in assembly, with no dependencies on other code and data. In contrast to the previous work, which focused on feasibility and performance, this work focuses on build system integration, code size, and security.

Tamas Ban [15] describes the motivation and design of code sharing between the first stage bootloader and the secure execution environment in Arm Trusted Firmware-M. He identified the potential to reduce the code size of TF-M for constrained Cortex-M microcontrollers by eliminating the duplication of cryptographic algorithms between MCUboot [2] and the MbedTLS PSA Crypto API [9] in the TF-M crypto service. This was achieved by providing the address of shared symbols via a linker script. The author claims a code size advantage from 1.2 kB up to 15 kB, depending on the configuration. However, the author mentions problems with Application Binary Interface (ABI) stability when performing firmware updates. In addition, the proposed solution only supports a limited selection of toolchains.

Boeckmann et al. [16] have worked on the deduplication of cryptographic primitives in the RIOT operating system. In their work, the authors describe the implementation and integration of the Arm PSA Crypto API [13] into RIOT. They show that by integrating different crypto modules for hardware accelerators and software libraries under a common API, significant code size reductions can be achieved. According to the authors, the code size of a sample IoT application was reduced from 50 kB to just 15 kB, highlighting the importance of code deduplication when it comes to crypto libraries in IoT applications.

3 Problem Statement

The goal of this work is to build the foundation of a secure firmware based on the method described in [27] and integrate it into RIOT. Design decisions must be made on how to integrate this secure firmware into the RIOT build process. For example, if the secure and normal execution environments were compiled and linked in separate build processes, this would result in each program having its own copy of libraries. This could result in a significant waste of ROM space, since code is duplicated across all environments. The opposite extreme, compiling and linking all code in the same build, could reduce memory footprint but also introduce security issues.

Given that RIOT already provides its own implementation of crypto algorithms, the questions arise whether it is possible to share this code with the secure firmware, what the obstacles are, and what possible solutions exist. To answer these questions, the next section will first provide an analysis of all code and libraries that could potentially be shared across execution environments. Each of these libraries will then be analyzed for functional issues and potential security issues. Other factors to consider are highlighted. Following this analysis, potential solutions to the issues raised are discussed. Based on these findings, the foundation of a secure firmware is then built and integrated into RIOT.

4 Analysis

4.1 Libraries

The first step in our analysis is to identify code or libraries in the RIOT code base that could potentially be shared with a secure firmware. To achieve this, the latest RIOT 2024.01 release was systematically evaluated. Mainly, all crypto modules in the system and package folders were listed. In addition, relevant system libraries of the GCC toolchain were selected. Based on this search, a selection of modules and libraries that are relevant to this discussion and potentially duplicated between execution environments are listed in Table 1.

Table 1: Libraries and RIOT Modules of Interest

Module / Library	**Functions of Interest**
sys/crypto	symmetric ciphers and message authentication codes
sys/hashes	cryptographic hashes
sys/random	pseudo random number generators
pkg/mbedtls	symmetric ciphers, asymmetric ciphers, digital signatures, cryptographic hashes, cryptographic random number generators
pkg/micro-ecc	elliptic curve digital signatures
libc	stdio: String formatting string: memcpy, memset, strlen, ...
libgcc	RISC-V save-restore routines

These modules and libraries were then further analyzed by building the RIOT examples/default application with all libraries included. Next, the generated object files were examined using binutils. This included the symbol tables and the generated assembly with the goal of identifying potential security vulnerabilities. The findings of the analysis are documented in the following sections.

4.2 Findings in Libraries

Denial of Service in sys/hashes

The RIOT hashes module provides implementations of common cryptographic hash algorithms such as MD5, SHA1, SHA224, SHA256, SHA512, and SHA3. These algorithms are also needed in trusted execution environments to support features such as secure boot and sealed storage. The symbol tables of the object files indicate that the implementation of the MD5, SHA1, SHA512, and SHA3 hash algorithms in the sys/hashes module are stateless because all code and data contained in the object file is read-only and have no side effects caused by external symbols. This makes the implementation of these algorithms suitable for sharing with the secure firmware.

The implementation of the SHA224 and SHA256 hash algorithms differ. The symbol tables of these algorithms reference a writeable variable called `PAD`, which is located in RAM. When cross-referencing the source code with the object file, as shown in Listing 1, it appears that this variable is used in the `sha2xx_pad` function which is called when finalizing SHA224 or SHA256 hashes.

Listing 1: RIOT SHA256 and SHA224 Implementation (Excerpt from sys/hashes/sha2xx_common.c in [3])

```c
static unsigned char PAD[64] = {
    0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
};
...
void sha2xx_pad(sha2xx_context_t *ctx)
{
    ...
    sha2xx_update(ctx, PAD, (size_t) plen);
    ...
}
```

It is a potential problem if the implementation is shared with the secure firmware. An attacker could use an out-of-bounds write in RIOT to change the value of the `PAD` variable. This would cause all hashes to be computed incorrectly, affecting not only RIOT, but all secure applications. After some investigation, it appears that this is a programming error. The `PAD` variable is incorrectly declared as non-const[1].

[1] A fix for this issue has been submitted to upstream RIOT and was merged with PR #20729 https://github.com/RIOT-OS/RIOT/pull/20729 (accessed 2024-06-21)

Information Disclosure in sys/random

The sys/random module in RIOT provides several implementations of pseudo random number generators. Random numbers are important for trusted execution environments to generate nonces for features such as remote attestation [8] and digital signatures [35]. By default, RIOT uses a linear congruential generator (LCG) [21]. Alternatively, a more secure SHA256-based pseudorandom number generator or hardware-based true random number generators can be selected.

The symbol tables of the sys/random object files reveal that both the LCG and the SHA256PRNG use global variables. A lookup of the symbols in the source code shows that these global variables also contain the state of the pseudo-random number generator. (See Listing 2)

Listing 2: RIOT Random Number Generator (Excerpt from sys/random/musl_lcg.c in [3])

```c
static uint64_t _seed;
uint32_t random_uint32(void)
{
    _seed = 6364136223846793005ULL*_seed + 1;
    return _seed>>32;
}
```

From a security perspective, this is a problem because pseudo-random number generators are deterministic, which would allow the normal execution environment to predict all future random numbers generated by secure applications. It could also be used to change the state to a known value controlled by the attacker. Therefore, it is not possible to share the PRNG implementation provided by RIOT. However, it is important to note that this is due to a design choice made by RIOT. If the API were stateless, like is done in MbedTLS (See Listing 3), where the caller passes the context of the PRNG to the library as a parameter, this would not be a problem.

Listing 3: MbedTLS Random Number Generator (Excerpt from include/mbedtls/ctr_drbg.h in [9])

```c
int mbedtls_ctr_drbg_random(void *p_rng,
                            unsigned char *output, size_t output_len);
```

Thread Safety in pkg/mbedlts

MbedTLS is a crypto library that can be optionally included as a package in RIOT. It provides an alternative implementation of cryptographic algorithms for ciphers, hashes, and signatures. It also provides an implementation of the TLS and DTLS protocols.

The object file generated by MbedTLS contains writable global variables. Analyzing the source code reveals that these variables are function pointers used for a platform-specific mutex implementation. When compiling MbedTLS for an RTOS like RIOT, the library relies on implementation-provided locking primitives to allow concurrent use by multiple preemptible threads [11].

Listing 4: MbedTLS Threading API (Excerpt from include/mbedtls/threading.h in [9])

```
void mbedtls_threading_set_alt(void (*mutex_init)(mbedtls_threading_mutex_t *),
                               void (*mutex_free)(mbedtls_threading_mutex_t *),
                               int (*mutex_lock)(mbedtls_threading_mutex_t *),
                               int (*mutex_unlock)(mbedtls_threading_mutex_t *));
extern void (*mbedtls_mutex_init)(mbedtls_threading_mutex_t *mutex);
extern void (*mbedtls_mutex_free)(mbedtls_threading_mutex_t *mutex);
extern int (*mbedtls_mutex_lock)(mbedtls_threading_mutex_t *mutex);
extern int (*mbedtls_mutex_unlock)(mbedtls_threading_mutex_t *mutex);
```

RIOT provides adapters to its own threading API via the interface shown in Listing 4. When RIOT boots, it initializes MbedTLS by calling the `mbedtls_threading_set_alt` function. This makes the library unsuitable for sharing with secure firmware because the secure execution environment cannot rely on the scheduler and mutex implementation of an application in the normal execution environment.

Control Flow Hijack and Privilege Escalation in pkg/micro-ecc

uECC [22] (micro-ecc) is a library that provides functionality for creating and verifying elliptic curve digital signatures. Trusted execution environments rely on digital signatures to implement features such as secure boot, keystores, remote attestation, and sealed storage. Inspection of the uECC object files generated by the RIOT build system shows that the uECC library is not stateless. The library contains a writeable symbol `g_rng_function` that is located in RAM. This symbol is used in the `uECC_sign` function and is part of the API shown in Listing 5.

Listing 5: uECC API (Excerpt from uECC.h in [22])

```
typedef int (*uECC_RNG_Function)(uint8_t *dest, unsigned size);
void uECC_set_rng(uECC_RNG_Function rng_function);
uECC_RNG_Function uECC_get_rng(void);
```

Because the RNG function is implemented as a function pointer in global memory, it is not suitable for shared use between the secure and normal execution environments for the functional reasons already mentioned in the previous section. But this is not the only problem. A shared function pointer in RAM can also be used to hijack control flow from the secure world. By changing the value of this variable, an attacker can cause a secure application to jump to a specific location in memory, resulting in privilege escalation.

System Calls and Constructors in libc

The C Standard Library (libc) is a common library used by many embedded applications. It includes ubiquitous functions from the standard string library such as memcpy, memset, and strcmp. It also includes the standard I/O library, which provides functions to format strings. RIOT supports two libc implementations: Newlib and picolibc. Both have the same problems when it comes to sharing code with the secure firmware. They implement system calls by statically linking against interfaces provided by the RIOT operating system. This approach has the advantage that the user can use standard functions such as open, read, write, and close to interact with the RIOT vfs subsystem, or use printf to write to the serial console. However, this makes it unsuitable for use by other applications in the secure execution environment who wish to use their own version of system calls.

More fundamental problems arise from the interactions between the C Standard Library and the toolchain. Languages like C and C++ provide mechanisms for executing code before the main function [24]. In C, functions marked with the `__attribute__((constructor))` decorator are placed in a special `.init_array` section of the object file. When the microcontroller boots, before the main function is called, all constructors are executed through the `__libc_init_array` function. A similar mechanism exists for globally constructed C++ objects. This causes problems when integrating multiple applications into the same build process, because not all constructors belong to the same execution environment. They should be called at different times in different contexts.

Shareable Libraries

Other libraries such as libgcc and sys/crypto (except for another chacha-based PRNG) do not hold global state and have no side effects. libgcc provides functionality such as RISC-V save/restore routines and FP emulation. sys/crypto provides implementations of symmetric cyphers such as AES and Chacha. These libraries can be shared between the normal and secure execution environments without modification.

4.3 Additional Considerations

Relative Addressing

The default for RIOT is to compile programs with linker relaxations enabled. Linker relaxations are used to reduce the code size of RISC-V binaries [30]. The RISC-V instruction set architecture currently specifies a 32-bit instruction format with 16-bit compressed instructions [33]. Addressing an arbitrary memory location on a 32-bit RISC-V microcontroller requires two 32-bit instructions or 8 bytes of code.

To reduce the amount of code needed to access global variables, RISC-V toolchains can reduce the code needed to access some 32-bit memory addresses by relaxing them against the global pointer. The RISC-V ABI [18] defines the general purpose register x3 as the global pointer. The value of this register is determined by the linker. The linker can then represent access to memory locations as a 7-bit offset from the gp register in only 2 bytes, or as a 12-bit offset in 4 bytes of code [31].

These gp-relative memory accesses are an obstacle when trying to share code between the secure and normal execution environments, because the gp register must have the same value for all code. You must also ensure that an attacker cannot change the value of the gp register at runtime. Otherwise, the integrity of the code could be compromised. Linker relaxations can be disabled in the toolchain [1].

Firmware Updates and Secure Boot

Trusted execution environments typically require the implementation of secure boot to ensure the integrity of the code running on a microcontroller [28]. Secure boot establishes a chain of trust from an immutable boot loader up to the code of the normal execution environment. This is achieved by having each boot stage verify the signature of the next code that is loaded and executed.

Depending on how the boot stages of the secure firmware are defined, this raises the question of how the shared code is signed and verified in the chain of trust. For example, in Arm Trusted Firmware-M, the Secure Processing Environment (SPE) and the Non-Secure Processing Environment (NSPE) can be signed separately [10]. While it may be reasonable to assume that a later boot stage can trust the code of an earlier boot stage, this makes the chain of trust more complicated and harder to verify.

It also complicates the installation of software updates. It may be necessary to update software in different execution environments independently of each other. For example, upgrading RIOT without upgrading the secure firmware. Some parts of the secure firmware may be immutable [7]. This can create problems when updating shared dependencies that contain API- or ABI-breaking changes.

TCB Complexity

Finally, the testing and verification of the secure firmware has to be taken into account to ensure the security and safety of systems that use it. A larger and more complex trusted computing base (TCB) makes it more difficult to verify its correctness [23]. For trusted execution environments, it is even desirable to formally verify parts of the firmware [28], which requires it to be small and of low complexity. Therefore, sharing code between secure and normal execution environments is opposed to formal verification and, in general, to the security of the secure firmware.

4.4 Solutions

The last section identified barriers to sharing code between RIOT and the secure firmware. This section discusses solutions to these problems. For simplicity, the previous findings can be further grouped into three categories:

1. Issues with global variables.
2. Issues with the toolchain and build process.
3. Issues with complexity and testing.

First, libraries or shared code that hold global state are vulnerable to information disclosure, denial of service, and control flow hijacking. This is because global variables are accessible from both normal and secure execution environments. Furthermore, sharing global state may also be undesirable from a functional perspective.

Second, some libraries are statically linked to interfaces provided by the implementation. Also, linking secure and normal applications in the same build causes problems with the toolchain regarding initialization and relative addressing. Even more problems are caused by software updates and the requirement to sign normal and secure applications separately.

Third and finally, there are security benefits to not sharing code between security-critical components, such as the secure monitor, and normal applications. For one thing, it makes it easier to apply auditing techniques such as testing and formal verification. And even without formal verification, the attack surface of the secure environment is reduced.

Dynamic Linking

A solution to the first problem, and partially to the second, might be dynamic linking. Dynamic linking is a technique for sharing code between applications in systems with virtual memory. It works by placing shared libraries in physical memory once, and then mapping those libraries into the virtual memory of each application in such a way that it has its own set of global variables. It is made possible by Position Independent Code (PIC), which is supported by the RISC-V ISA [18]. However, it requires the presence of an MMU, which is not present on most microcontrollers and is not supported by the RIOT operating system.

Park et al. [26] describe a method for implementing shared libraries for microcontrollers without the need for virtual memory. Their technique allows shared libraries to be placed at fixed locations in ROM. The paper shows a significant 35 % reduction in flash size with only a 4 % performance penalty. However, it requires modifications to the toolchain.

PSA APIs

Another approach can be seen in Arm Trusted Firmware-M. Arm has solved the duplication of cryptographic algorithms in its TF-M solution for Cortex-M microcontrollers using Remote Procedure Calls (RPCs). The Arm PSA architecture defines a common

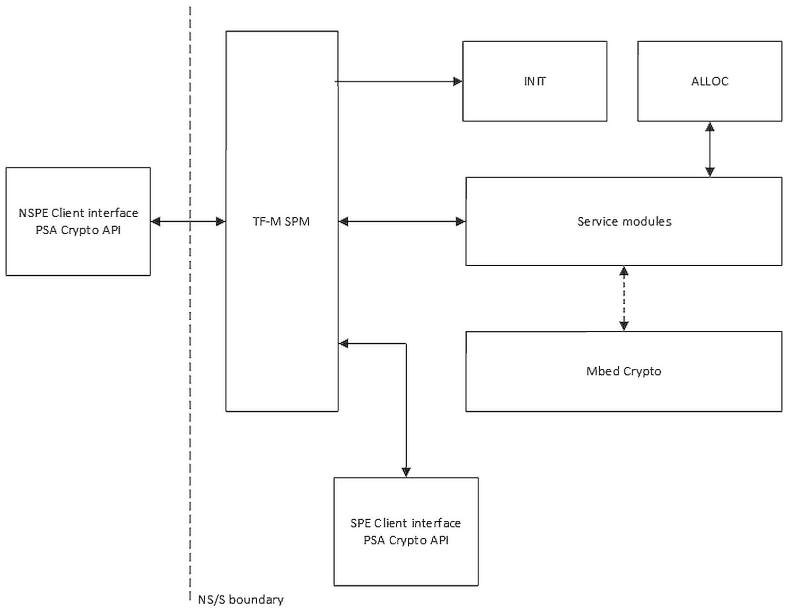

Figure 1: PSA Crypto Service in Arm TF-M (Image Source: [6], Visually Enhanced)

PSA Crypto API [13] that provides generic interfaces for symmetric and asymmetric cryptography. This API is then implemented by a crypto service within the secure environment. An application in the normal environment and enclaves are then compiled against stubs of the PSA Crypto API as shown in Figure 1.

The stubs (Client Interface) intercept calls to the PSA Crypto API, then package the parameters into I/O vectors and make a system call to invoke the TF-M crypto service. This service then unpacks the parameters and calls the PSA Crypto implementation of the MbedTLS library contained in the crypto service. After the request is completed, the results are returned to the stub, which unpacks the results and returns them to the caller.

The Arm approach is a very simple solution to the code duplication problem. However, there are a few minor problems. First, not all software is built and ready to use the PSA Crypto API. Second, the generic design of the PSA Crypto API makes it very complex to implement. Both the PSA implementation in MbedTLS and RIOT contain over 20 k LOC. Problems have been found with PSA in MbedTLS [29]. Third, Arm does not provide numbers on the performance overhead of the RPC. And finally, the approach is incompatible with RIOT, which provides its own implementation of the PSA Crypto API [16].

A solution to the last problem might be to intercept calls to the crypto libraries at the driver level instead of calls to the PSA Crypto API itself. This way, the implementation of PSA Crypto can reside inside RIOT. Calling the PSA backends will invoke the crypto library of the secure firmware.

4.5 Discussion

This research began with the goal of finding ways to reduce the duplication of code shared by applications in the secure and normal execution environments. An analysis showed that there are many security-related problems when trying to share code between security-critical and insecure applications. Some solutions to these problems can be found in both the literature and in practice.

Ultimately, there is a trade-off between code size and code complexity. Given the experimental status of dynamic linking in microcontrollers, it is clear that sharing code across isolation boundaries is not a good idea at this time. Therefore, a more traditional approach of linking each application against its own set of libraries is the only reasonable choice at this time. Code size reduction of crypto libraries can be achieved using the PSA Crypto API.

5 Implementation

Based on the results of the previous analysis, this section describes the implementation and integration of a secure firmware into RIOT. The firmware is developed for the SiFive FE310 microcontroller [32]. It is based on the same method for running RIOT in RISC-V user mode described in [27]. However, it has been completely rewritten to meet the new requirements.

In the prototype developed in [27], the code of both the machine mode secure firmware and the user mode RTOS were interleaved in the same repository. They were both compiled and linked by the RIOT build process and then flashed to the microcontroller as a unified binary.

Additionally, an experimental system call interface and PSA Crypto API driver are developed to allow RIOT to use cryptographic algorithms implemented by the secure firmware. The code size advantages of using the approach are evaluated for the P256R1 elliptic curve implementation of micro-ecc.

5.1 Requirements

From the above analysis, the following requirements should be met:

1. Separate build processes
   The secure firmware and RIOT should be built independently of each other.
2. No direct code sharing
   The secure firmware and RIOT should not be linked against shared library code.
3. Independent updates
   The secure firmware and RIOT can be updated independently.

Additionally, the following properties are desirable:

5. Easy to enable
   The secure firmware should require a single option to enable.
6. Memory footprint
   The memory usage should be reduced to fit constrained IoT devices.

5.2 Challenges

Separating the secure firmware and RIOT into their own repositories and build processes introduces a few new challenges that must be addressed. First, there is a need for well-defined interfaces between the two pieces of software. This is a problem because there is a requirement not to share code between execution environments. Some mechanism must be found to share the interface definitions.

Second, both execution environments must agree on a memory layout. Both the secure firmware and RIOT require their own memory space. Consequently, they must be assigned disjoint address ranges in RAM and ROM. The problem with this is how to allocate these address ranges when the build processes are separated.

5.3 Solution

5.3.1 Secure Firmware

For the secure firmware, a new repository and build process is created. The repository contains all the source code that runs in machine mode on the microcontroller. This code includes:

- RAM initialization:
  Loading initialized data, clearing zero initialized data.
- Memory protection:
  Initialize PMP to protect code and data of the secure firmware.
- Interrupt delegation to user mode.
- Exception handling.
- System calls for access to machine mode CSRs from user mode.
- Elliptic curve cryptography:
  The micro-ecc library and system calls to sign and verify signatures.

The secure firmware uses makefiles as a build system. The toolchain chosen is the xPack RISC-V GCC [5] compiler for reproducible builds and the picolibc [25] standard C library. The firmware repository contains the definitions of the user API as well as the definitions of the memory layout. The memory layout is designed so that the firmware is placed at the beginning of the ROM and RAM, and RIOT is placed after the firmware.

When the firmware is built, the following artifacts are generated:

- Secure firmware image
- User API headers
- Linker script variables, containing the address range for the user mode RTOS

These artifacts can then be tested and audited and are then packaged for use in RIOT.

5.3.2 RIOT Integration

These artifacts are made available to the RIOT build system as a package. In order to make the secure firmware usable within RIOT, a number of specific changes have been made to the RIOT code base. They include:

- Conditionally adjust the start address for ROM and RAM in the linker script.
- Conditional changes to the `riscv_common` CPU target to allow interaction between RIOT and the secure firmware.
- Addition of Makefile targets to flash the secure firmware.

To accomplish this, a new pseudo module `riscv_user` has been added. This module is used in combination with the `IS_USED` macro to enable any changes in the RIOT code base required when the user chooses to use the secure firmware.

To flash both secure firmware and RIOT using the RIOT build system, an approach similar to RIOT bootloaders is taken. RIOT already has the ability to flash bootloaders and RIOT independently to support software updates. These bootloaders are riotboot [4] and MCUboot [2]. riotboot essentially provides the Makefile targets `riotboot/flash-bootloader`, `riotboot/flash-slot1`, and `riotboot/flash-slot2` to build and flash the bootloader and images independently. It also provides a more convenient Makefile target called `riotboot/flash`, which flashes the bootloader and RIOT in one command. MCUboot takes a simpler approach than riotboot, providing a make target `mcuboot-flash-bootloader` and a target named `mcuboot-flash` that flashes the bootloader and RIOT at the same time.

Inspired by both the riotboot and MCUboot make commands, the following make targets have been defined for the secure firmware:

- `riscv_user/flash-bootloader`
  Flashes the secure firmware onto the microcontroller.
- `riscv_user/flash-riot`
  Flashes the RIOT app onto the microcontroller.
- `flash`
  Flashes both the secure firmware and the RIOT app onto the microcontroller.

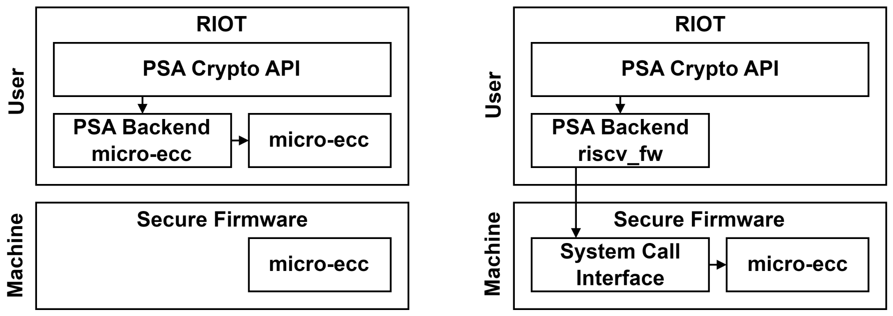

Figure 2: Deduplication of the micro-ecc Library Using the PSA Crypto API

In addition, the flash dependencies of the RIOT build system have been set up so that invoking the RIOT `flash` target causes both the bootloader and RIOT to be built and flashed.

5.3.3 PSA Crypto API

Within RIOT, a driver for the PSA Crypto API is implemented that can call the micro-ecc library residing in the secure firmware. micro-ecc was chosen for better comparisons because it is the default ECC library used by the PSA Crypto implementation in RIOT.

Figure 2 illustrates how code deduplication works with the PSA Crypto API. On the left, you can see how the micro-ecc library is duplicated across isolation boundaries. Both RIOT and the secure firmware contain their version of the micro-ecc library. This duplication can be avoided by using a custom PSA ECC backend. On the right, a PSA driver that interfaces with the secure firmware via system calls can use the micro-ecc library included in the secure firmware. This eliminates the need to include the micro-ecc library in RIOT, resulting in a potentially significant reduction in code size and increased protection of the micro-ecc code and state variables.

5.4 Evaluation

Functional Tests

The newly written secure firmware was evaluated in several ways. First, the secure firmware was functionally tested using the RIOT test suite. More specific, the RIOT core, timer, and PSA subsystem tests were run with the secure firmware. All tests pass.

Code Size

Second, the code and data size of the secure firmware and RIOT were analyzed. These numbers are not very meaningful at this time because the firmware only implements interrupt delegation, system calls, startup code, and the micro-ecc library. It does not yet implement features such as key stores or remote attestation. However, these numbers do provide a baseline for what to expect.

Table 2: Code Size of the Secure Firmware

Section	**Size in ELF [Bytes]**	**Effective Size [Bytes]**
ROM	6590	8192
RAM	1160	2048
.data	0	-
.bss	136	-
.stack	1024	-

Table 2 shows that the compiled secure firmware requires 6590 bytes of ROM and 1160 bytes of RAM. An additional 36 bytes of RAM is reserved for shared data between RIOT and the secure firmware, bringing the effective size to 1196 bytes. The ROM size must be rounded up to 8 KiB because the flash on the SiFive HiFive 1B board can only be erased with 4 KiB granularity [20]. The RAM must also be rounded up to account for additional RAM space required by future updates.

Table 3 shows a comparison of the code and data size of RIOT when compiled with the micro-ecc PSA backend compared to the secure firmware PSA backend. The ROM size was reduced by 6448 bytes, a reduction of almost 30 %. RAM usage is identical.

Usability

The usability of the new firmware is optimal in the sense that it can be enabled with a single pseudo module in RIOT. Listing 6 shows a comparison of a command used to build and flash a RIOT application with and without the secure firmware.

Table 3: Code Size of RIOT with and without Code Deduplication

Section	**Size w/ PSA micro-ecc Backend [Bytes]**	**Size w/ PSA riscv_user Backend [Bytes]**
ROM	23086	16638
.init	102	102
.text	21416	15148
.rodata	1568	1388
RAM	5940	5940
.data	212	212
.bss	5472	5472
.stack	256	256

Listing 6: Comparison of Commands to Build and Flash a RIOT Application

```
make all flash
USEMODULE=riscv_user make all flash
```

Maintainability

Finally, maintainability has been greatly improved over the previous prototype in [27]. The previous work defined separate targets for `riscv_common_user`, `fe310_user`, and `hifive1b_user`, resulting in significant code duplication within RIOT. The new firmware uses five conditional changes to the existing `riscv_common` CPU target in the startup code, trap handler, and riscv_encodings. This means that any changes to `riscv_common` do not need to be applied to other targets, greatly improving maintainability.

6 Conclusion and Outlook

The implementation of the secure firmware can be considered a success. While it is suboptimal that some libraries are duplicated between RIOT and the secure firmware, the deduplication of crypto libraries, using micro-ecc as an example, via the PSA Crypto API has shown significant code size benefits. With this result, the applications and secure firmware will most likely fit within the 100 KiB limit set by RFC7228.

The implementation provided will serve as a solid foundation for future development of features such as key stores, sealed storage, remote attestation, and more. Further research should be done on the security of the secure firmware. This includes non-functional and security testing, as well as exploration of formal verification techniques.

References

[1] *GCC Documentation: RISC-V Options.* – URL https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html. – Accessed 2024-05-10

[2] *MCUboot.* – URL https://github.com/mcu-tools/mcuboot. – Accessed 2024-05-10

[3] *RIOT - The friendly Operating System for the IoT.* – URL https://github.com/RIOT-OS/RIOT. – Accessed 2024-05-10

[4] *RIOT Documentation: riotboot.* – URL https://api.riot-os.org/group__bootloader__riotboot.html. – Accessed 2024-05-10

[5] *The xPack GNU RISC-V Embedded GCC.* – URL https://xpack.github.io/dev-tools/riscv-none-elf-gcc/. – Accessed 2024-05-10

[6] Angelis, Antonio de: *Crypto Service design.* – URL https://tf-m-user-guide.trustedfirmware.org/design_docs/services/tfm_crypto_design.html. – Accessed 2024-05-10

[7] Arm Limited: *BL1 Immutable bootloader.* – URL https://tf-m-user-guide.trustedfirmware.org/design_docs/booting/bl1.html. – Accessed 2024-05-10

[8] Arm Limited: *Initial Attestation Service Integration Guide.* – URL https://tf-m-user-guide.trustedfirmware.org/integration_guide/services/tfm_attestation_integration_guide.html. – Accessed 2024-05-10

[9] Arm Limited: *MbedTLS.* – URL https://github.com/Mbed-TLS/mbedtls. – Accessed 2024-05-10

[10] ARM LIMITED: *Secure boot.* – URL https://tf-m-user-guide.trustedfirmware.org/design_docs/booting/tfm_secure_boot.html. – Accessed 2024-05-10

[11] ARM LIMITED: *Thread safety and multithreading: concurrency issues.* – URL https://mbed-tls.readthedocs.io/en/latest/kb/development/thread-safety-and-multi-threading/. – Accessed 2024-05-10

[12] ARM LIMITED: Arm TrustZone Technology for the Armv8-M Architecture, Version 2.1. (2018). – URL https://developer.arm.com/documentation/100690/0201/?lang=en. – Accessed 2024-05-10

[13] ARM LIMITED: *PSA Crypto API 1.1.* Februar 2022. – URL https://developer.arm.com/documentation/ihi0086/latest/. – Accessed 2024-05-10

[14] BACCELLI, Emmanuel ; GÜNDOĞAN, Cenk ; HAHM, Oliver ; KIETZMANN, Peter ; LENDERS, Martine S. ; PETERSEN, Hauke ; SCHLEISER, Kaspar ; SCHMIDT, Thomas C. ; WÄHLISCH, Matthias: RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. In: *IEEE Internet of Things Journal* 5 (2018), Nr. 6, S. 4428–4440. – URL https://doi.org/10.1109/JIOT.2018.2815038

[15] BAN, Tamas: *Code sharing between independently linked XIP binaries.* Mai 2020. – URL https://tf-m-user-guide.trustedfirmware.org/design_docs/software/code_sharing.html. – Accessed 2024-05-10

[16] BOECKMANN, Lena ; KIETZMANN, Peter ; LANZIERI, Leandro ; SCHMIDT, Thomas C. ; WÄHLISCH, Matthias: Usable Security for an IoT OS: Integrating the Zoo of Embedded Crypto Components Below a Common API. In: *Proc. of Embedded Wireless Systems and Networks (EWSN'22).* New York, USA : ACM, October 2022, S. 84–95. – URL https://dl.acm.org/doi/10.5555/3578948.3578956

[17] BORMANN, Carsten ; ERSUE, Mehmet ; KERÄNEN, Ari: *Terminology for Constrained-Node Networks.* RFC 7228. Mai 2014 (Request for Comments). – URL https://www.rfc-editor.org/info/rfc7228. – Accessed 2024-05-10

[18] CHENG, Kito ; CLARKE, Jessica: *RISC-V ABIs Specification, Document Version 1.0.* November 2022. – URL https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/tag/v1.0. – Accessed 2024-05-10

[19] Hex Five Security, Inc.: *Hex Five MultiZone Security Datasheet*. 2020. – URL https://hex-five.com/wp-content/uploads/2020/01/multizone-datasheet-20200109.pdf. – Accessed 2024-05-10

[20] Integrated Silicon Solution, Inc.: *IS25LP032D IS25WP032D*. August 2023. – URL https://www.issi.com/WW/pdf/25LP-WP032D.pdf. – Accessed 2024-05-10

[21] Knuth, Donald E.: *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd. USA : Addison-Wesley, 1997. – ISBN 0201896842

[22] MacKay, Ken: *micro-ecc*. – URL https://github.com/kmackay/micro-ecc. – Accessed 2024-05-10

[23] McCune, Jonathan M. ; Parno, Bryan ; Perrig, Adrian ; Reiter, Michael K. ; Seshadri, Arvind: Minimal TCB Code Execution. In: *2007 IEEE Symposium on Security and Privacy (SP '07)*, URL https://doi.org/10.1109/SP.2007.27, 2007, S. 267–272

[24] Packard, Keith: *Initializers/Constructors and Finalizers/Destructors in Picolibc*. – URL https://github.com/picolibc/picolibc/blob/main/doc/init.md. – Accessed 2024-05-10

[25] Packard, Keith: *picolibc - a C library designed for embedded 32- and 64- bit systems.*. – URL https://github.com/picolibc/picolibc. – Accessed 2024-05-10

[26] Park, Jiyong ; Lee, Jaesoo ; Kim, Saehwa ; Hong, Seongsoo: Quasistatic shared libraries and XIP for memory footprint reduction in MMU-less embedded systems. In: *ACM Trans. Embed. Comput. Syst.* 8 (2009), jan, Nr. 1. – URL https://doi.org/10.1145/1457246.1457252. – ISSN 1539-9087

[27] Pfau, Lars: Measuring the Performance Overhead of RIOT Running in RISC-V User-Mode. (2024), Februar. – URL https://inet.haw-hamburg.de/teaching/ws-2023-24/project-class/pr1_lars_pfau.pdf. – Accessed 2024-05-10

[28] Sabt, Mohamed ; Achemlal, Mohammed ; Bouabdallah, Abdelmadjid: Trusted Execution Environment: What It is, and What It is Not. In: *2015 IEEE Trustcom/BigDataSE/ISPA* Bd. 1, 2015, S. 57–64

[29] SHIN, SeongHan ; OGAWA, Tomoyuki ; FUJITA, Ryo ; ITOH, Mari ; YOSHIDA, Hirotaka: An Investigation of PSA Certified. In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. New York, NY, USA : Association for Computing Machinery, 2022 (ARES '22). – URL https://doi.org/10.1145/3538969.3544452. – ISBN 9781450396707

[30] SIFIVE: *All Aboard, Part 3: Linker Relaxation in the RISC-V Toolchain*. August 2017. – URL https://www.sifive.com/blog/all-aboard-part-3-linker-relaxation-in-riscv-toolchain. – Accessed 2024-05-10

[31] SIFIVE: *All Aboard, Part 4: The RISC-V Code Models*. September 2017. – URL https://www.sifive.com/blog/all-aboard-part-4-risc-v-code-models. – Accessed 2024-05-10

[32] SIFIVE INC.: *SiFive FE310-G002 Manual v1p5*. September 2022. – URL https://sifive.cdn.prismic.io/sifive/034760b5-ac6a-4b1c-911c-f4148bb2c4a5_fe310-g002-v1p5.pdf. – Accessed 2024-05-10

[33] WATERMAN, Andrew ; ASANOVIĆ, Krste: *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. Dezember 2019. – URL https://github.com/riscv/riscv-isa-manual/releases/tag/Ratified-IMAFDQC. – Accessed 2024-05-10

[34] WATERMAN, Andrew ; ASANOVIĆ, Krste ; HAUSER, John: *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*. Dezember 2021. – URL https://github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12. – Accessed 2024-05-10

[35] WEISER, Samuel ; SCHRAMMEL, David ; BODNER, Lukas ; SPREITZER, Raphael: Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations. In: *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, August 2020, S. 1767–1784. – URL https://www.usenix.org/conference/usenixsecurity20/presentation/weiser. – ISBN 978-1-939133-17-5