Troubles in embedded porting land? An investigation using Rust and Zig for IoT device security

Your N. Here
Your Institution

Second Name
Second Institution

Abstract

Many Internet of Things (IoT) devices use C to program them; but C can be dangerous. Newer systems-languages like *Rust* and *Zig* enable the development of embedded systems with stronger security guarantees: but are they usable by developers familiar with C? We ported a flash-based secure store from C to both Rust and Zig; and find that, while it is possible to employ these languages for embedded development; usability hurdles remain for developers to overcome. Furthermore, using these languages does not necessarily provide any additional security guarantees.

1 Introduction

Internet of Things (IoT) devices have a reputation for being insecure [3]. Many run on restricted platforms with limited computing power and a small amount of memory [5]. Often programmed using C or C++, newer languages like *Rust* [6] and *Zig* [9] holds the promise of enhancing security, offering more up-to-date tools for developers, and integrating seamlessly into existing development projects. But are these languages usable by embedded *C* and *C++* systems developers to gain security when porting existing embedded projects?

Our poster explores our reflections in porting a Secure Non-Volatile Store (NVS)—suitable for holding secrets on a device—to an IoT device, and the usability challenges faced. Future work will expand this to a developer study to explore the challenges these languages face more generally when porting to embedded devices.

USENIX Symposium on Usable Privacy and Security (SOUPS) 2026.
August 23–36, 2026, Hannover, Germany.

2 Porting method

Background work was undertaken before starting the port, including reading the language documentation and doing several exercises to become familiar with each language's concepts and syntax. To test the utility of each language, an existing library, *Secure NVS* [2], was ported to each language. The library was chosen because:

- It has a well-defined Application Programming interface (API) that follows the C calling convention.
- It uses a hardware abstractions layer (HAL) to isolate the implementation from the specific hardware. The HAL can remain in C, reducing the amount of code that needs porting.
- It uses both packed structures and bit fields to manipulate memory structures [4], which is typical in embedded applications but more unusual in broader programming.

Each C function was first syntactically converted to the target language, and then was reviewed and converted to follow the idiomatic patterns for each language, such as error handling, optional parameters and memory management. The design was then reviewed to follow language patterns, such as object construction patterns or the use of pointers. The C code was also updated to use a consistent naming convention for functions, allowing us to track a function from C to Rust or Zig. As each new function was ported, unit tests were added in each language. The porting was completed in one month, with both the Rust and Zig implementations written in parallel by a single developer.

The developer's notes and reflections on the porting process are presented in this poster to capture the challenges they faced when porting the application. They were an experienced embedded systems engineer with roughly 15 years of commercial C and C++ development experience. Future work will explore whether their experience generalises with a qualitative developer study.

Table 1: Implementation details for each port of the NVS program.

		C	Rust	Zig
Lines of code		4,763	4,155	2,933
Runtime (s)	Debug	2.06	3.82	2.57
	Release	2.07	1.95	2.03
	ReleaseSmall	–	2.25	1.94
	ReleaseFast	–	–	1.96
Compiled size (KB)	Debug	332	23,846	2,268
	Release	26	22,262	2,474
	ReleaseSmall	–	7,545	24
	ReleaseFast	–	–	273

3 Reflections

Table 1 shows the lines of code for each implementation alongside various benchmarks. The solution was built for both debug and release, and a benchmark test was executed for each build. The C code was compiled with `-O2` using `gcc 15.2.0`. Runtimes presented are the averages over 100 runs. Rust code was compiled using `rustc 1.95.0`. Rust generates very large library images; an optimised release version was created using best practice guidelines, which is shown as *ReleaseSmall*. For Zig (`zig 0.14.1`), the release benchmark is the output for *ReleaseSafe* optimiser, as this is the suggested optimiser for release code.

3.1 Rust

The borrowing and ownership model is a key feature of Rust. Converting functions from C highlighted issues when trying to conform to the model. In general, this was not as problematic as expected. In some cases, the restrictions did require the developer to jump through additional hoops. Rust does not support bit fields; the getter/setter pattern was used to do the bit field manipulation manually. Accessing members of a packed structure also required the developer to manage the way packed values are accessed. The C NVS library uses the technique of taking the address of the packed structure and then writing the content into flash memory. For Rust, casting a pointer in this way requires an unsafe block, making much of the implementation unsafe.

The C NVS library is memory aware, making all memory allocation at startup, and using multiple lists of pointers to the resources that it tracks. The Rust implementation required significant refactoring and the addition of an object abstraction layer to manage this functionality. Much of this additional work felt unnecessary as the code was designed to manage the memory requirements optimally.

Building a Rust implementation for an embedded environment requires developers to forgo using the standard library, i.e. building the source with the `[no_std]` directive. Most of the documentation and example code use the standard library, making it difficult to find example code that can be used.

3.2 Zig

Zig provides the functionality to import a C include file, which then generates the Zig equivalent definitions, making the integration with existing C code straightforward. Zig supports both packed structures and bit fields. However, there are issues in the way this is implemented, making working with packed structures problematic. See the example Zig code below:

```
// The &self.size returns the address of the struct,
// not the address of the member,
// so we do it manually!!
const kv_hddr_size_buf = @as([*]u8,
         @ptrFromInt(@intFromPtr(self) +
         @offsetOf(KvHddr,"size")))
         [0 .. @bitSizeOf(KvSize) / 8];
```

Cross-compiling is a first-class use case for the Zig compiler, making it trivial to build on one environment for deployment on embedded systems. The Zig compiler is also a drop-in replacement for the clang or `gcc` C compilers. The Zig build environment can be used to build Zig and C projects. It would be complicated to move big C projects from `Cmake` to use the Zig build system.

4 Conclusions

Both Rust and Zig support embedded environments and can be used to replace existing C implementations. While porting to Zig was a less intrusive process — Zig is more like C, but with better error handling and resource management — porting to Rust did require redesign; the porting process was more complicated and time-consuming. The resulting code for both languages is also significantly larger, which is an issue when porting to a restricted resource IoT device, as memory quotas are *absurdly* low [1, 7, 8]: bytes matter and whilst the *ReleaseSmall* Zig optimiser was lower than the C's binary size.

Rust feels like a more mature language with lots of third-party libraries that provide additional functionality. but also forces the developer to think about resource management at compile time. Zig is a newer language with less support and documentation. Both languages provide benefits over using standard C, with more robust resource management and better error handling. Zig error handling felt less intrusive and more natural as a C developer.

For a developer accustomed to working with C, Zig provides a smoother porting transition, requiring less redesign. However, both languages need to do more to support embedded developers.

References

[1] Parvaneh Asghari, Amir Masoud Rahmani, and Hamid Haj Seyyed Javadi. Internet of Things applications: A systematic review. *Computer Networks*, 148:241–261, January 2019.

[2] Andrew Baldrian. libnvs - github, 2025.

[3] Mario Ballano Barcena and Candid Wueest. Insecurity in the Internet of Things. Technical report, Security response, Symantec, 2015.

[4] GNU. Top (GNU C Language Manual), 2017.

[5] Abhishek Khanna and Sanmeet Kaur. Internet of Things (IoT), Applications and Challenges: A Comprehensive Review. *Wireless Personal Communications*, 114(2):1687–1762, September 2020.

[6] Steve Klabnik, Carol Nichols, and Chris Krycho. *The Rust Programming Language - The Rust Programming Language*. rust-lang.org, February 2025.

[7] Tammy Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Newnes, December 2012. Google-Books-ID: 96jSXetmlzYC.

[8] Vishal A. Thakor, Mohammad Abdur Razzaque, and Muhammad R. A. Khandaker. Lightweight Cryptography Algorithms for Resource-Constrained IoT Devices: A Review, Comparison and Research Opportunities. *IEEE Access*, 9:28177–28193, 2021.

[9] ziglang.org. *Documentation - The Zig Programming Language*. ziglang.org, 2025.

A Source code examples

A.1 C

```
/**
 * @brief encrypt the kv size and append the kv header to the flash
 *
 * @note the program buffer must be open so that we can append this data
 *
 * @param[in] self    The header of the key value
 * @param[in] prog_buf The program buffer to amend the data to
 * @param[in] crypt   The crypt wrpper
 * @param[in] crypt_ctx The encryption context to decrypt the key and value
 *
 * @return NVS_ERR_OK    - The operation was successful
 *         NVS_ERR_HAL_BASE - Errors defined by the flash HAL
 */
static nvs_err_t nvs_KvHddr_append_enc(
    const nvs_kv_hddr_t *self,
    nvs_prog_buf_t *prog_buf,
    const nvs_crypt_t *crypt,
    nvs_crypt_ctx_t *crypt_ctx)
{
    assert(self != NULL);
    assert(prog_buf != NULL);
    assert(nvs_ProgBuf_is_open(prog_buf));
    assert(crypt != NULL);

    // we are only encrypting the header size, not the full header
    // so we need to do this first and then append the header
    nvs_kv_size_t kv_size_encrypted;
    nvs_Crypt_encrypt_part(crypt, crypt_ctx, (uint8_t *)&(self->size),
        sizeof(nvs_kv_size_t), (uint8_t *)&kv_size_encrypted);
    // append the unencrypted part of the header
    nvs_err_t err = nvs_ProgBuf_append(prog_buf, sizeof(nvs_kv_hddr_t)
        - sizeof(nvs_kv_size_t), (uint8_t *)self);
    if (err != NVS_ERR_OK)
        return err;
    // now append the encrypted size
    return nvs_ProgBuf_append(prog_buf, sizeof(nvs_kv_size_t), (uint8_t *)&kv_size_encrypted);
}
```

A.2 Rust

```
/// encrypt the kv size and append the kv header to the flash
///
/// The program buffer must be open so that we can append this data
///
/// 'self'    The header of the key value
/// `prog_buf` The program buffer to amend the data to
/// `crypt`   The crypt wrapper
/// `ctx`     The crypt context to use to encrypt /decrypt the key and value
fn append_enc(
    &self,
    prog_bug: &mut ProgBuf,
    crypt: &Crypt,
    ctx: &mut nvs_crypt_ctx_t,
) -> Result<(), NvsError> {
    let mut kv_size_encrypted = [0 as u8; mem::size_of::<KvSize>()];
    crypt.encrypt_part(
        ctx,
        unsafe {
            slice::from_raw_parts(
                &self.size as *const _ as *const u8,
                mem::size_of::<KvSize>(),
```

```
            )
        },
        kv_size_encrypted.as_mut_slice(),
    )?;
    // append the unencrypted part of the header
    prog_bug.append(unsafe {
        slice::from_raw_parts(
            self as *const _ as *const u8,
            mem::size_of::<KvHddr>() - mem::size_of::<KvSize>(),
        )
    })?;
    // now append the encrypted size
    prog_bug.append(kv_size_encrypted.as_slice())?;
    Ok(())
}
```

A.3 Zig

```
/// Encrypt the kv size and append the kv header to the flash
///
/// The flash buffer must be open so that we can append this data
///
/// `self`    The header of the key value
/// `prog_buf` The program buffer to amend the data to
/// `crypt`   The crypt wrapper
/// `ctx`     The encryption context to decrypt the key and value
fn append_enc(
    self: *const KvHddr,
    prog_buf: *ProgBuf,
    crypt: *const Crypt,
    ctx: *c.nvs_crypt_ctx_t,
) ProgBuf.AppendError!void {
    // we are only encrypting the head size, not the full header
    // so we need to do this first and then append the header

    // the &self.size returns the address of the struct,
    // not the address of the member, so we do it manually!!
    const kv_hddr_size_buf = @as([*]u8, @ptrFromInt(@intFromPtr(self)
        + @offsetOf(KvHddr, "size")))[0 .. @bitSizeOf(KvSize) / 8];
    var kv_size_encrypted: [@bitSizeOf(KvSize) / 8]u8 = undefined;
    try crypt.encrypt_part(ctx, kv_hddr_size_buf, &kv_size_encrypted);

    // append the unencrypted part of the headder
    try prog_buf.append(@as([*]u8, @ptrCast(@constCast(self)))
        [0 .. (@bitSizeOf(KvHddr) - @bitSizeOf(KvSize)) / 8]);
    // now append the encrypted size
    try prog_buf.append(&kv_size_encrypted);
}
```