embeddedworld2022
Exhibition&Conference
... it's a smarter world

Rust for Secure IoT Applications

Why C Is Getting Rusty

Mario Noseda, Fabian Frei, Andreas Rüst, Simon Künzli
Zurich University of Applied Sciences (ZHAW)
Institute of Embedded Systems (InES)
Winterthur, Switzerland
mario.noseda@zhaw.ch, fabian.frei@zhaw.ch, andreas.ruest@zhaw.ch, simon.kuenzli@zhaw.ch

***Abstract*— Memory corruption is still the most used type of exploit in today's malware landscape. Human error inevitably introduces memory vulnerabilities into software by using memory-unsafe languages like C and C++, affecting not only security but also safety, dependability, and even basic functionality of devices. The Rust language guarantees memory safety without a garbage collector, promises comparable performance to C/C++, and allows for gradual extension of existing codebases by using its foreign function interface. This report presents the risks of having memory vulnerabilities in embedded applications, what a switch to Rust looks like, how the development experiences differ between Rust and C/C++, and if there are significant differences in performance.**

Keywords*— *bare-metal programming; c; cpp; cybersecurity; embedded systems; exploits; iot; memory unsafety; rust

I. INTRODUCTION

Embedded and bare-metal programming environments predominantly use C and C++, which is not surprising due to the balanced mix between hardware-oriented programming and high-level abstraction. Although these languages have clear and undeniable strengths, they also have weaknesses that can no longer be overlooked with today's software development know-how and experience from the last decades.

A. Memory Unsafety

Arguably the most critical of these weaknesses is the inherent memory unsafety [1], a property of programming languages that allows bugs to arise from incorrect memory use. These vulnerabilities can be divided into two categories: spatial and temporal vulnerabilities. Typical spatial vulnerabilities are out-of-bounds array accesses, dereferencing a null pointer, and using uninitialized memory. Regarding temporal vulnerabilities, use-after-free, double-free, and data races are known problems (these lists are not exhaustive). Since the advent of malware in the 1980s [2], exploits have been using memory vulnerabilities extensively because they usually allow a severe compromise of the target. Fish in a Barrel's iOS14 analysis in 2021 identified 346 vulnerabilities, of which 209 (60%) were due to memory unsafety [3]. Among actively exploited vulnerabilities, they attributed 8 out of 11 (73%) to memory unsafety. Microsoft's Security Response Center says that memory unsafety caused 70% of their assigned CVEs [4]. The Google Security Blog even speaks of a memory unsafety share of 90% of all Android bugs, mainly consisting of out-of-bound reads and writes, use-after-free, and integer overflows [5]. Finally, Google's Project Zero analyses 0-day exploits and concluded that 39 of 58 exploits in 2021 are due to memory corruption [6]. So, although the problem of memory unsafety has been known for a long time, decades of experience have not led to the production of software without critical memory vulnerabilities today. On the contrary, according to the sources mentioned above, memory unsafety is the most common cause of vulnerabilities. We will not be able to produce bug-free code anytime soon without significantly changing software development.

B. Problem for Everyone

Even if one does not care about security, memory unsafety also has a tremendous impact on the basic functionality of a device. Memory bugs can be extremely hard to find and can stay undetected by various static and dynamic code analysis tools, even worse if they only appear after a specific run time in the field. Bugs like this are notoriously difficult to reproduce and trace, which makes troubleshooting enormously difficult and, in the worst case, can result in a significant loss (both monetary and reputational). Memory unsafety is arguably just as crucial for those who deal with systems and devices where safety and dependability are essential. There will hardly be any device that does not require either security, safety, dependability, or just basic functionality. Memory safety should therefore be an ideal to strive for, no matter what domain you come from or what demands the product has, as it ties into all of them. Of course, no one claims that memory safety means you will no longer have other types of bugs in your code. It only eliminates memory bugs and does not protect the developer from introducing other bugs, such as logic errors, into the code.

However, especially concerning embedded devices and the Internet of Things (IoT), getting rid of a whole category of bugs is extremely valuable. It reduces the probability of a critically necessary update, which can be extremely costly in the case of a large number of devices. This becomes even worse if the devices do not have sufficient connectivity, requiring the update to be performed manually, maybe even onsite.

C. What Can We Do About It?

There are only two options for achieving memory safety. Either we use various tools and tests to check code written in a memory-unsafe language for bugs and vulnerabilities, or we use a memory-safe language like Rust from the beginning. This report contains insights and experiences from embedded software developers who, after more than a decade with C, have decided to evaluate what a switch to (embedded) Rust looks like and how it compares to working with C/C++. On the one hand, the report summarizes publicly available information, which will be of interest to all those who want to take this step without digging through the vast amounts of documentation. Additionally, the report also contains findings on how the development experiences differ between Rust and C/C++.

This report is structured accordingly: Section II introduces Rust and how the ecosystem and the software development workflow differ from C/C++. Further, it lists drawbacks and how one can achieve memory safety without using Rust. Lastly, the section discusses which companies and projects already employ Rust. Section III investigates the use and benefits of Rust in Embedded Systems. Section IV presents a proof of concept (PoC) application written in C, the possible damages caused by exploitation, and an equivalent but memory-safe PoC application written in Rust. Section V compares the performance of the languages with a benchmark consisting of cryptographic primitives, and section VI draws appropriate conclusions.

II. RUST

Rust is a compiled, strong- and statically-typed systems programming language with high-level features while retaining low-level memory management capabilities. Graydon Hoare started developing Rust at Mozilla Research in 2010. Mozilla has gradually reduced its influence on the project while open-source contributions steadily increased [7]. Rust is now stewarded by the Rust Foundation since its formation in February 2021 [8].

According to Steve Klabnik (former member of the Rust core team and co-author of the official Rust book) [9], Rust has memory safety, speed, and productivity as its core values. Ergonomics, compilation time, and correctness are seen as secondary values. Regarding the last point, it is essential to note that Rust strongly cares about your program being correct, but just not in the sense that it would force you to use dependent types or a proof assistant. However, Rust does not create new programming paradigms and uses previously established ones. Further, it does not release unfinished features only for the sake of releasing and will not guarantee support for old and obsolete targets.

So far, it sounds like Rust could be just a younger and less mature version of C++. However, it boasts memory safety without the need for a runtime and garbage collector, which is the case for common memory-safe languages like Java, C#, or Python. It achieves this with the concept of ownership, Rust's arguably most unique feature. It is responsible for most memory safety guarantees and prevents memory bugs with either a compilation error or a controlled panic during run time. Table I shows how various issues are handled by Rust.

TABLE I: TYPICAL MEMORY BUGS AND WHEN RUST HANDLES THEM.

Issue	Rust (release)
Out-of-bounds R/W	Run time
Null dereference	Run time[1]
Type confusion	Run time[1]
Integer overflow	Run time[1]
Use-after-free	Compile time
Double free	Compile time
Invalid stack R/W	Compile time
Uninitialized memory	Compile time
Data race	Compile time

1: Some restrictions apply

A. Rust's Improved Ecosystem

Memory safety is not the only new or improved feature compared to C/C++. Working with Rust, you quickly realize that Rust is a much newer language and was designed with the flaws and shortcomings of C/C++ in mind. For example, switching the RTOS of an application might take multiple days just to set up the C/C++ build system by installing all the dependencies and the required toolchain. Many of these build systems are also famously convoluted, making this even harder. Installing Rust is extremely simple with the "rustup" toolchain installer [10], even when cross-compiling for bare-metal targets. And Rust's internal package manager, "Cargo" [11], takes care of downloading dependencies, compiling your package, and even distributing it if you wish to do so. Furthermore, Rust contains various language features typically associated with high-level or scripting languages like pattern matching, verbose backtraces, or generic functions and types (similar to C++ templates). Such features are unusual for languages that run on bare-metal targets. This section will not go into detail on what Rust can and cannot do but rather show how these features had a positive impact on the development experience.

As mentioned before, installing Rust is extremely easy, and you will not have to fight with some obscure build system as everything has been standardized with Cargo. Next up is learning the actual language itself. The community prides itself on enforcing an open and inclusive environment for newbies and veterans alike. The creation and maintenance of learning resources that are needed to get into the language are usually led by members of the core team with contributions from the community. Notably, the superb documentation of the language and associated software keeps the barrier low for new contributors.

B. How Rust Development Differs From C/C++

This section uses a fictional application development to discuss how the development experiences of C/C++ and Rust differ. Imagine you are starting the project and want to take your first steps with the library you will use. Many C libraries pack related data into structs, which need to be set up and configured using specific functions before they can be passed to multiple functions by reference to achieve some arbitrary tasks. Listing I

illustrates some potential function prototypes for drafting and approving a post.

```
typedef struct {
    ...
} post_t;

int post_init(post_t *post);
int post_add_content(post_t *post, uint8_t *content);
int post_request_review(post_t *post);
int post_approve(post_t *post);
```

LISTING I: C FUNCTION PROTOTYPES FOR DRAFTING A POST.

C's type system does not assist newcomers to the library, as all functions just need a pointer to the struct without encoding further information into the types. The user needs to make sure that they call the required functions in the correct order, and finding all of them might be even harder. Contrastingly, Rust's type system wants the library's author to encode as much information into the types as possible. Listing II demonstrates what the corresponding Rust function prototypes could look like.

```
fn new_draft_post(content: String) -> DraftPost {...}
fn request_review(draft: DraftPost) -> PendingReview {...}
fn approve_post(pend: PendingReview) -> Post {...}
```

LISTING II: RUST FUNCTION PROTOTYPES FOR DRAFTING A POST.

Thus, the type system guides the user automatically to all the necessary functions as many of the types can only be produced by calling the respective functions instead of creating them directly. In the case of the example, the user can only create values of the type `Post` by calling the listed functions in order. Of course, a C library could also specify different struct types for the different stages of the post. However, this is neither the convention nor can C prevent the user from simply creating an instance of the final post struct directly and skipping the functions altogether.

Assume that you progressed with your fictional application development and are happily working with Rust's enums. There is a similar concept in other languages called "tagged unions", as they allow data of different types and sizes to be attached to the enum variants. Listing III illustrates an enum for storing either an IPv4 address using four 8-bit integers, an IPv6 address as a string, or no address without any associated data. Rust checks the type during run time, and thus the data cannot be interpreted for another variant.

```
enum IpAddr {
    V4(u8,u8,u8,u8),
    V6(String),
    None,
}

let ipv4 = IpAddr::V4(127,0,0,1);
let ipv6 = IpAddr::V6(String::from("::1"));
let no_addr = IpAddr::None;
```

LISTING III: ENUM EXAMPLE IN RUST.

Listing IV shows how this could be implemented in C using a struct and an enum.

```
typedef enum {
    V4,
    V6,
} ip_version_t;

typedef struct {
    ip_version_t version;
    uint8_t *addr;
} ip_addr_t;

uint8_t addr_buf[4] = {127,0,0,1};
ip_addr_t ipv4 = {
    .version = V4,
    .addr = addr_buf,
};
```

LISTING IV: ENUM EXAMPLE IN C.

Rust's enums are not just significantly more concise but also safe compared to C's alternative, as setting and checking the current variant is mandated by the syntax. In contrast, C allows the developer to forget to set or check the enum variant. Interpreting the address field for the wrong variant or a similar mistake would go undetected and result in undefined behavior. This boils down to the same problem as forgetting the null pointer check in C when supplying functions with pointers.

Unfortunately, even good code is not perfect, and errors will likely happen during run time. Most C libraries follow the convention to return an integer corresponding to the occurred error. This approach, however, does not work well with bubbling up the error to the upper layers without loss of information or significant effort to implement sophisticated error handling. Rust again uses the "tagged union" functionality of their enums by defining result values that allow functions to return data with different types and sizes in case of success or an error. Listing V shows how a function can return an integer if successful or a string in case of an error.

```
fn foo(a: u8, b: u32) -> Result<u32,String> {
    ...
}
```

LISTING V: RUST FUNCTION USING A RESULT TYPE AS A RETURN VALUE.

Of course, C could define a tagged union for returning different types from a function as well. However, this introduces the same problem of forgetting to check which variant is currently stored in the union, as discussed in the previous enum example. Apart from that, returning tagged unions in C is also somewhat uncommon.

Rust has an excellent testing infrastructure and encourages the developer to add corresponding unit tests directly under the code-under-test. The tests are typically excluded during building to keep the executable as small as possible. Using the `cargo test` command, they can be executed whenever desired. Listing VI demonstrates the recommended practice of placing Rust code and its corresponding unit test into the same file.

```rust
fn foo(a: u8, b: u32) -> Result<u32,String> {
    ...
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_foo() {
        assert_eq!(foo(1,1000).unwrap(), 123);
    }
}
```

LISTING VI: FUNCTION AND ITS CORRESPONDING UNIT TEST.

Assume that all tests have passed in your fictional application development, your application reached version 1.0.0, and you shipped it to the customer. Even though you thoroughly tested your application, you still missed something, and the client immediately encountered a bug. You quickly find the reason, fix the bug, build the application, and run the tests. Astonishingly, the unit tests pass, but the integration tests fail. It appears that the change to the code was correct as the unit tests passed. However, you added a so-called doc comment to the function in question. As implied by the name, doc comments allow the developer to add documentation to code elements like functions, methods, or types. Rust highly encourages developers to add example code to such doc comments as the compiler converts them to integration tests. Listing VII shows a function with a doc comment containing sample code. Executing the first line of code in the doc comment verifies if calling the function works as intended, while the second line checks the return value similar to a unit test.

```rust
/// Add two numbers.
///
/// # Example
/// ```
/// let res = add(2,3);
/// assert_eq!(res, 5);
/// ```
fn add(a: u32, b: u32) -> u32 {
    ...
}
```

LISTING VII: FUNCTION WITH DOC COMMENT CONTAINING EXAMPLE CODE.

Regarding the fictional application development, you modified the function signature and its call sites. However, you forgot to update the description and the example in the doc comment. Fortunately, the outdated example code resulted in an integration test failure which prompted you to update the entire doc comment. Outdated doc comments can be very problematic as they might lead to others misusing your code. Similar functionality can be found in other languages like Python [12] and Haskell [13]. Due to the integration test failure, you could correct the documentation, and you can use Cargo to create the project documentation. It aggregates all doc comments and publishes the resulting documentation as an interlinked webpage for easy navigation, which is one of the reasons why published Rust crates (libraries) are so well documented.

The features discussed in this section, Rust's type system, improved error handling, testing infrastructure, along with many other features, and its safety guarantees result in the developer being able to focus more on the logic and the functionality of the application instead of constantly worrying about introducing bugs into the code. This significantly increases productivity and confidence in the resulting product.

C. *Drawbacks*

Of course, do not trust anyone that claims that Rust does not have any real drawbacks. For example, there is currently no clear definition of what features a programming language needs to qualify as object-oriented. Rust does not make any claims [14] but argues to be somewhere in between as it allows the creation of objects (structs and enums) that package data and define corresponding procedures (methods) that act on this associated data. Another common feature is the encapsulation of the data in these objects, which Rust is also capable of (private and public attributes). Lastly, inheritance is also widely regarded as a feature of object-oriented languages, which Rust intentionally does not support [15]. Rust uses a different approach (traits and trait objects) instead of inheritance to solve similar problems. A trait specifies one or more methods that need to be implemented by the type (sometimes called interface in other languages). If two types implement the same traits, they share the same behavior. However, this specific take on object orientation adds to the complexity of the language as it is yet another concept that either has to be learned from scratch or one has to adjust to if coming from a language like C++.

Furthermore, the concept of ownership introducing memory safety is entirely new for most programmers. The associated compile-time errors might appear complex and unfair at times as the compiler just relentlessly flags various lines of code if it detects the possibility of a memory safety violation. You can consider every C warning a Rust error and probably will get even more errors on top of that. Ultimately, the learning curve is very subjective, but we experienced C as easy to learn and hard to master and Rust as hard to learn and easy to master.

It is not a surprise that a language many times younger than C does not have the same ecosystem yet. Even though it is growing every day, there might not be a library for your specific use case yet. Moreover, although Rust has full Windows support for the compiler and the standard library, the community is decidedly Linux-focused, resulting in some third-party libraries supporting Windows only partially or not at all.

Various design choices of Rust require that the compiler has access to the source code of the application as well as all the dependencies written in Rust. This requirement effectively makes it impossible to publish compiled Rust libraries which might be a problem for companies wanting to preserve their intellectual property. However, Rust's licensing model is permissive enough for releasing your source code with a commercial license that requires anyone to buy it first. If the source code must be kept secret at all costs, a Rust library can also be compiled to a static library with a C interface, indistinguishable from a static C library. However, this results in the Rust compiler not being able to check security guarantees across the entire application for anyone that uses this library. The application itself, as well as the library, are checked individually,

but the calls to and from the library cannot be checked, as the C interface of the static Rust library effectively acts as a barrier to the safety checks of the compiler. Thus, the developer needs to check the safety and correctness of the calls to and from the library. Fortunately, this is possible with reasonable effort if the API of the library is clearly defined and manageably sized. Furthermore, suppose a memory vulnerability still manages to be detected. You can limit the search to the interaction between the application and the library, making the search much more accessible, especially for large projects. Lastly, the current rise of open-source culture and this characteristic of the Rust compiler may lead to more vendors releasing their source code. This would significantly improve the debugging experience compared to working with precompiled libraries, which are effectively a black box for the end-user.

Another often-debated drawback is Rust's compilation time, as compiling a large project depending on various other crates can take some time to compile for the first time. Type checking, turning generic into specific implementations, running the borrow checker (this is what verifies the ownership rules), and compiling all third-party crates from source just takes time. However, this is time that you would otherwise spend debugging when using a memory-unsafe language, i.e., you "pay" for the safety guarantees with an extended compilation time. Luckily, subsequent builds only compile the changed crates, which results in very short compilation times in most use cases.

Lastly, the currently missing compiler certification might be a dealbreaker for some industry sectors. Fortunately, the German company Ferrous Systems, which is already involved in multiple Rust open-source projects and working groups [16], takes on the challenge of providing a Rust compiler toolchain version qualified for ISO26262 [17][18], which is a functional safety standard for electrical and electronic systems in series production passenger cars. They plan on tackling other certifications if this project is successful.

D. Alternative Solutions

Instead of using a memory-safe language, one could also keep using C/C++ and use specialized tools to ensure memory safety. Unfortunately, this is only treating the symptoms instead of the root cause, which shows as soon as one analyzes their efficiency in regard to finding memory bugs. Static code analysis (SCA) is only as good as the ruleset, which must be finetuned to whatever it is applied. Moreover, as soon as a library is used without access to the source code, the SCA has no choice but to assume that calls to and from such libraries, as well as the library itself, are correct. This returns the responsibility to the developer, which is precisely the opposite of what the SCA tried to achieve in the first place.

Most SCA tools parse the code into an abstract representation which is then analyzed. This analysis is path sensitive as bugs might only occur if various code segments are executed in a specific order. However, this poses a significant problem for SCA as a full exploration of all paths quickly exhausts the provided resources. As most software developers do not have access to a supercomputer, the tool must make assumptions on what probably will not happen during run time or other approximations. This situation is further exacerbated if asynchronous paths like interrupts, exceptions, or concurrency can occur, with many tools simply ignoring the possibilities [19]. In contrast, Rust again uses the sophisticated type system, which annotates every type if it is safe to be sent over a thread boundary and if it contains the required synchronization mechanics (e.g., mutex) for safe accesses from different contexts. In other words, Rust forces you to take all necessary precautions while providing concurrency guarantees at compile time instead of run time.

Dynamic code analysis (DCA) is often used together with SCA, as these two approaches complement each other. Unfortunately, both combined are still not able to detect all types of memory bugs. This is further complicated because DCA can only find bugs that actually occur during a test run, and memory bugs are known to be very elusive. Additionally, these run-time checks can impose a significant performance impact, resulting in the tests not being run under the same conditions as the device-under-test. This does not even consider that many DCA checks are only available for x86 and other desktop-style architectures. For a Cortex-M architecture, they would need to be ported manually, or the firmware could be checked within an emulation. Though, this approach would require mocking the used peripherals and maybe even externally connected devices in case of networking. In contrast, everything that Rust cannot catch at compile time (like indexing an array with a number supplied by the user) is handled with implicitly added run-time checks that trigger a controlled panic for keeping the system in a known and controlled state. These checks could be regarded as mandatory DCA while resulting in minimal to zero overhead, depending on the situation. The default panic handler mainly prints a backtrace to the standard error output and aborts the program. This behavior is usually infeasible for embedded systems due to missing OS abstractions or because simply aborting the program and requiring a user to restart the application is impossible (e.g., pacemaker). Thus, custom panic handlers can be registered to select the desired behavior in case of run-time errors, like resetting the target. Similar to conventional watchdogs triggered by anomalous execution times and deadlocks, this is effectively a watchdog triggered by run-time bugs.

Other tools to identify bugs include fuzzing and emulation: Fuzzing can be used to verify the input of functions, but other than that, it is severely limited in its use cases. Emulation might also help find some more bugs, but it is usually extremely costly to set up the environment, create all the needed mocks, and keep everything up to date.

To test how thoroughly the discussed SCA/DCA tools find bugs, we created 16 small C/C++ applications containing different types of memory bugs. We used popular tools like splint (out-of-the-box configuration), cppcheck (using the MISRA C 2012 ruleset), and GNU C/C++ sanitize for our tests. Listing VIII shows one of the 16 applications leading to a memory safety issue.

```
void foo(uint32_t x) {
    uint32_t buf [10];
    if (x == 100) {
        buf[x] = 0;
    }
}
```

LISTING VIII: EXAMPLE MEMORY VULNERABILITY BUG.

Table II shows if the different tools flag the bugs correctly. Of course, one could improve the configuration of the tools so that they might find more bugs. However, this is a clear advantage for Rust as it executes all tests every time without requiring manual configuration and is able to find all these bugs. Such an approach eliminates the inevitable human errors associated with the configuration of such software checks.

TABLE II: CHECKING IF SCA/DCA TOOLS FIND MEMORY BUGS.

Bug	cppcheck	splint	GNU C/C++ sanitize
0	✓	✓	✓
1	✓	✓	✓
2	✓	✓	✓
3	missed	✓	missed
4	missed	✓	compile err.
5	✓	✓	missed
6	✓	✓	missed
7	missed	✓	missed
8	✓	✓	partly
9	partly	✓	partly
10	✓	missed	missed
11	missed	✓	missed
12	✓	✓	missed
13	missed	✓	partly
14	✓	incompatible	✓
15	missed	incompatible	missed

Instead of using tools to make an unsafe language safe, one could also use a different memory-safe language than Rust, like Go, C#, Java, Swift, Python, or JavaScript. However, getting these languages to run at a comparable performance on an embedded device (if at all!) would be a significant challenge, to say the least. Moreover, the language Zig [20] is not entirely memory-safe, but it presents itself as a better C alternative and provides significantly more checks than C. However, Table III shows that it is still not close to the safety guarantees of Rust [21].

TABLE III: C, ZIG, AND RUST MEMORY SAFETY COMPARISON.

Issue	Zig (release-safe)	Rust (release)
Out-of-bounds R/W	Run time	Run time
Null dereference	Run time[1]	Run time[1]
Type confusion	Run time[1,2]	Run time[1]
Integer overflow	Run time	Run time[1]
Use-after-free	None[1]	Compile time
Double free	None[1]	Compile time
Invalid stack R/W	None	Compile time
Uninit. memory	None	Compile time
Data race	None	Compile time

1: Some restrictions apply, 2: Partial

Furthermore, Zig is even younger than Rust as it started in 2016 and is currently on version 0.9.1 as of the release of this report.

E. Where Is Rust Used?

Not every company clearly states what kind of languages they use in their products. However, one can safely assume that members of the Rust Foundation use the language in some of their products or production workflow. The platinum members AWS, Google, Huawei, Meta, Microsoft, and Mozilla, are huge companies with the necessary budget to make such drastic changes as introducing a new programming language into products. Therefore, it is probably more interesting to look at the silver members like 1password, Arm, Dropbox, and Threema as they better represent all the businesses besides the tech giants.

The Internet Security Research Group (ISRG) [22] started the Prossimo project [23] to move the Internet's security-sensitive software infrastructure to memory-safe code. They depend on an active community of developers and funders to reach their goals. Probably the most famous of their initiatives is the journey to get Rust as a language for writing Linux Kernel drivers [24]. They created a branch of the Linux Kernel and currently apply necessary changes to facilitate the use of Rust. Even though Linus Torvalds voiced concerns in his reply to the RFC, he gave his implicit approval by not shutting it down immediately (unlike his infamous reaction to C++ [25]), which makes Rust the first language besides C to be considered anywhere near the Linux Kernel. Rustls is another one of their initiatives and is an alternative to OpenSSL written in Rust. It passed its first audit in 2020 and is ready to be used according to the maintainers. Furthermore, they have currently active initiatives to create memory-safe implementations in Rust for the network time protocol (NTP), domain name system (DNS) resolvers, curl (ubiquitous network transfer utility), and mod_tls (used in HTTP server by Apache).

III. EMBEDDED RUST

Generally, Rust supports both desktop PCs and embedded systems because the language allows complete low-level access. Nevertheless, the Embedded Rust Book [26] clearly distinguishes the following embedded programming classifications: hosted and bare-metal environments. The former is close to a desktop environment as it provides the application with a system interface like POSIX for accessing systems like the file system, threading, or networking. The latter does not include OS abstractions or other code that runs before the actual application. Rust's standard library (libstd) is implicitly added to all applications and requires OS abstractions like those provided in a hosted environment. Thus, the libstd is not available in bare-metal environments. However, Rust's core library (libcore) is a platform-agnostic subset of libstd, allowing the creation of applications for bare-metal environments without the need for any OS abstractions.

A. Portability of Embedded Drivers

Diving deeper into bare-metal environments, interacting with core peripherals usually either requires writing a custom driver and accessing the registers directly or using the drivers supplied by the manufacturers. Both approaches are very bad for portability if you want to change the MCU family (or even

manufacturer) down the line. The embedded working group is developing the embedded-hal crate [27] to build an ecosystem of platform-agnostic drivers. They specify traits for various peripherals (like ADC, SPI, Timers, …) and authors of drivers for a specific peripheral on a specific target are then encouraged to design their drivers using these traits. This allows applications to switch devices as all the calls to the peripherals are the same if the drivers all use the embedded-hal. The embedded-hal is gaining significant traction and represents a strong step towards increased portability of bare-metal applications.

B. *The Type System and Peripherals*

Even though the type system is already great as it is, it is even more helpful when dealing with the setup of hardware resources like peripherals or GPIOs. For example, see the function signature of `uart_init(…)` in Listing IX for a fictional initialization function of a UART driver. For the TX pin, the developer then needs to supply the function with a variable of the type `GpioOutput` to satisfy the type system and get an instance to control the UART. However, the GPIO driver, which declared this type, does not simply allow the creation of such a variable. It forces the developer to use the appropriate function to initialize an output pin from an uninitialized pin, which returns the required type (see example function signature `gpio_init_output(…)` in Listing IX). Of course, this procedure applies to the RX pin and various other peripherals that require setting up the hardware as well. Encoding the pin state like this ensures that misconfiguration (or simply forgetting to initialize) is an issue of the past [28].

```
fn uart_init(baudrate: u32,
             tx: GpioOutput,
             rx: GpioInput) -> Uart {...}

fn gpio_init_output(pin: GpioUninit) -> GpioOutput {...}
```

LISTING IX: ENCODING PIN STATES INTO TYPES IN RUST.

Notably, truly idiomatic Rust would further use its object-oriented characteristics to tie the data types and methods together even more.

C. *C and Rust Coexisting*

Finally, imagine an IoT device being developed using an RTOS written in C. Depending on the development philosophy of a company, it might be fair to assume that the developers working on the business logic are more likely to introduce bugs than the underlying RTOS, which is probably getting tested more thoroughly and has more eyes on the code itself. Of course, bugs will inevitably come up in the RTOS as it is unlikely that they will never make a mistake when using a memory-unsafe language. However, a device's probability of a critical bug can be significantly reduced if at least the business logic on top is written in a safe language, which usually offers the most considerable bug potential. Especially if the device manufacturer is understaffed, on a budget, and pressed with deadlines.

IV. PROOF OF CONCEPT

CVE-2020-6007 [29] is a heap-based buffer overflow vulnerability in Philips Hue Bridge that allows the attackers to install malicious firmware on smart light bulbs and spread it to other IoT devices within the same network. This vulnerability was the catalyst for this evaluation and report.

A. *PoC in C*

We created a proof of concept (PoC), which is loosely based on the CVE. It consists of an nRF52840 (Cortex-M4) [30] running a C application built on top of the Zephyr RTOS [31], which simply prints strings received over an open UDP server. Instead of using ZigBee, which is used by Hue products, we used OpenThread [32] as it is built on top of the same physical layer (IEEE 802.15.4 radio) and was already known to the authors. We deliberately designed the application in C with a simple memory vulnerability that can be in or excluded during compile time. Listing X shows the (very artificial) vulnerability in question, which is a simple `memcpy(…)` instruction using the source length without clamping it to the maximum size of the destination, resulting in a typical buffer overflow.

```
void vulnerable_print(const char* input, int len)
{
    static int count = 0;
    char buffer[16];

    /* Missing checks of input arguments */

    /* Copying into buffer w/o checking for overflows */
    memcpy(buffer, input, len);
    buffer[len] = 0;

    printk("[%u]: %s\n", count++, buffer);
}
```

LISTING X: SIMPLE MEMCPY() VULNERABILITY.

B. *Buffer Overflow Exploits*

Notably, C code compiled for the ARMv7E-M architecture used in the Cortex-M4 stores the base pointer of the last frame and the return address to the calling function in the first two words of the stack frame, as displayed in Figure 1.

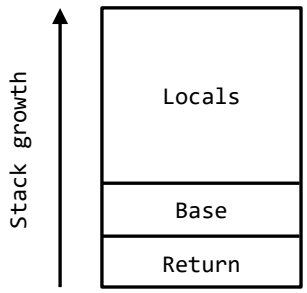

FIGURE 1: TYPICAL STACK FRAME LAYOUT.

This vulnerability thus enables the attacker to call some other function already present in the target's flash (like opening the safe or outputting some sensitive data) by overwriting the return address with the address of the target function. Of course, exploiting this either requires knowledge of the firmware or brute-forcing the target address. Listing XI shows what such an exploit could look like.

```
|<-- Filling buffer with dummy data -->|<-Ret->|
11111111222222223333333344444444555555550d040000
```

LISTING XI: EXPLOIT FOR REDIRECTING EXECUTION FLOW.

Instead of filling the buffer with dummy data and overwriting the stored return address, the attacker could also fill the buffer with shellcode (malicious machine code manually crafted for the target architecture) and overwrite the return address with the address of the buffer itself. This instructs the MCU to fetch its next instructions from the vulnerable buffer that the attacker just filled with shellcode. Listing XII illustrates an exploit containing shellcode.

```
|<--           Assembly code        -->|<-Ret->|
4ff0a0434ff4f032c3f81825c3f80c2555555555f1ef0020
```

LISTING XII: EXPLOIT USING SHELLCODE.

It is called "shellcode", as it is typically used to spawn a shell from which the attacker can control the target. However, it could also contain any other instructions like accessing a peripheral or exfiltrating sensitive data. Even worse is the fact that Cortex-M devices cannot make use of virtual addresses because of the missing memory management unit (MMU). Thus, established shellcode countermeasures like address space layout randomization (ASLR) are unavailable. Furthermore, there is no separation of kernel and userspace as seen in desktop environments, which would otherwise limit the reach of the exploits. At least most Cortex-M devices include a memory protection unit (MPU) that prevents unauthorized memory access. This countermeasure effectively prevents the attack, as the whole RAM (where the vulnerable buffer inside the stack is located) could be set to "not executable". It triggers an exception as soon as the address of the buffer is loaded into the instruction register at the beginning of the attack. Unfortunately, a study by W. Zhou et al. found the MPU of Cortex-M devices to be insufficient and circumventable [33]. According to the study, recent updates mitigated some of the issues, but the overall MPU design remains flawed.

Even if a functional and sophisticated MPU were present, an attacker could resort to return-oriented programming (ROP) if they have access to the target firmware. This exploitation technique allows arbitrary code execution without injecting it into the target system by misusing code already present in its flash memory. Conventional countermeasures are either unavailable on Cortex-M devices due to the missing hardware (e.g., MMU) or add significant overhead to the execution, making them infeasible for IoT's performance and energy constraints.

C. PoC in Rust

Rewriting the same application in Rust showed that it is possible to use a preexisting RTOS written in C and add one's business logic on top using Rust. Rust's promises for a zero-overhead foreign function interface (FFI) held up, as we could call Rust functions from C and vice versa. However, there was a clear limitation to FFI that arose during the development of our Rust PoC. C symbols, macros, and other preprocessor functionalities are not available from Rust as the FFI works during link-time. Specifically, the Zephyr RTOS used for this PoC uses macros for various things that are now unavailable from Rust. Depending on the macro, this could either be solved by packing it into a function or by creating a corresponding Rust function that contains the same functionality.

In contrast to the vulnerable C function in Listing X, idiomatic Rust does not pass a type containing contiguous memory (e.g., arrays) and the length of the stored data in this memory separately, as this is known to be error-prone. Instead, developers are encouraged to use a type that owns the memory and keeps track of the metadata. For example, the `String` type is actually a struct containing a pointer to the memory that holds the contents of the string, a field for storing the maximum capacity, and a field for the current length of the stored string. The fields are private and thus force the developer to use the `String`'s methods which keep track of the metadata and prevent memory violations.

Therefore, it was impossible to recreate the vulnerable function of Listing X in Rust, which is not surprising regarding Rust's memory safety guarantees. We created example functions for demonstrating how trying to misuse arrays results in either compile-time errors or run-time panics. Listing XIII attempts to copy an array of length 10 into an array of length 5, resulting in a compile-time error.

```
fn wrong_buf_len() {
    let dest = [0;5];
    let src = [0;10];
    dest = src;
}

error[E0308]: mismatched types
  --> src/main.rs:18:9
   |
18 |     dest = src;
   |            ^^^^ expected an array with a fixed size
                     of 5 elements, found one with 10
                     elements
```

LISTING XIII: COPYING AN ARRAY OF LENGTH 10 INTO AN ARRAY OF LENGTH 5 AND THE CORRESPONDING COMPILER ERROR.

Listing XIV uses a constant out-of-bounds index which also yields a compile-time error. This detection also works with arithmetic expressions as the index, as long as the compiler can evaluate them at compile time.

```
fn const_index_out_of_bounds() {
    let mut buf = [0;5];
    buf[10] += 1;
}

error: this operation will panic at runtime
  --> src/main.rs:23:2
   |
23 |     buf[10] += 1;
   |     ^^^^^^^ index out of bounds: the length is 5 but
                 the index is 10
```

LISTING XIV: USING OUT-OF-BOUNDS INDEX KNOWN AT COMPILE TIME AND THE CORRESPONDING COMPILER ERROR.

Listing XV shows the panic message printed during run time if the compiler cannot determine the index at compile time (e.g., supplied by the user).

```
thread 'main' panicked at 'index out of bounds:
the len is 5 but the index is 10', src/main.rs:28:5
```

LISTING XV: RUN-TIME PANIC MESSAGE WHEN ACCESSING THE 10TH ELEMENT OF AN ARRAY OF LENGTH 5.

The compiler can detect most memory vulnerabilities during compile time, similar to the shown examples. However, as the compiler cannot predict user input, run-time panics effectively prevent vulnerabilities from being exploited if the developer forgets to add the necessary checks.

As discussed in section III.C, the risk of a small team of developers adding a critical bug to their application is far greater than the RTOS having some critical vulnerability if it has a large community, is actively used, and is getting tested meticulously. Even though the entire application is not memory-safe by combining Rust and C, it significantly improves the confidence in the product without having to rewrite everything in Rust. This PoC indicates that Rust and C can coexist to improve IoT devices' stability, safety, and security.

V. CRYPTO BENCHMARK

The PoC was designed to evaluate the feasibility of Rust for embedded systems and how the development differs from C/C++. However, it is not suited for making statements about other metrics like execution time and memory footprint due to its limited scope. The former is especially important for battery-powered devices as execution time is usually directly related to battery life. Therefore, a crypto benchmark was created with hashing (SHA256), encrypting (AES-CCM, AES-GCM, CHACHA20-POLY1305), and decrypting using MbedTLS (C) [34] and RustCrypto (Rust) [35]. This provides a practical comparison as these are arguably the most used libraries of their respective language for bare-metal environments and both have been audited by third parties.

A. Results

The results listed in Table IV show that there seems to be no correlation between execution time and language selection.

TABLE IV: RELATIVE DIFFERENCE IN EXECUTION TIME FOR CRYPTOGRAPHIC ALGORITHMS WHEN SWITCHING FROM MBEDTLS (C) TO RUSTCRYPTO (RUST).

Algorithm	From C to Rust
SHA256 (16 B)	- 13 %
SHA256 (64 KiB)	- 9 %
AES128-CCM (16 B)	+ 145 %
AES128-CCM (64 KiB)	+ 73 %
AES128-GCM (16 B)	+ 101 %
AES128-GCM (64 KiB)	+ 20 %
CHACHA20-POLY1305 (16 B)	- 53 %
CHACHA20-POLY1305 (64 KiB)	- 52 %

Assumably, the implementation of the algorithms and the efficiency of the libraries themselves are more important for the resulting execution time. For example, the RustCrypto implementation of SHA256 completely unrolled all for loops which is probably responsible for the faster execution. However, this also results in a far larger memory footprint which is not always desirable or feasible.

B. The Computer Language Benchmarks Game

The Computer Language Benchmarks Game [36] uses ten simple but computationally expensive problems for benchmarking a large number of languages. Everyone can submit their optimized solutions, and many programmers use this as a challenge to squeeze every last bit of performance out of the benchmark of their favorite language. C, C++, and Rust share the podium on all of these benchmarks. We have awarded each language with points according to the achieved ranks: three points for 1st, two for 2nd, and one for 3rd place. The resulting scoreboard in Table V shows that these languages should be indistinguishable for most developers and use cases in regard to theoretical performance.

TABLE V: SCOREBOARD FOR THE COMPUTER LANGUAGE BENCHMARKS GAME.

Language	Points
C++	21
Rust	20
C	19

VI. CONCLUSIONS

Memory safety is essential for ensuring the security, safety, dependability, and even basic functionality of devices. Exploiting memory vulnerabilities easily leads to a fully compromised device, as conventional countermeasures are infeasible for most embedded systems. Our analysis, as well as today's vulnerabilities and 0-day exploits, indicate that tools like static and dynamic code analysis provide only inadequate protection, as even the tech giants with almost unlimited resources still struggle with memory safety. Rust not only prevents memory bugs due to its memory safety guarantees but also significantly improves the development experience compared to C/C++. Furthermore, no significant difference in performance could be detected, which is especially important for battery-powered devices. Of course, Rust's advantages and challenges compared to C/C++ must be weighed carefully before switching languages. However, having C and Rust coexist in an application allows for a successive transition without starting from scratch.

ACKNOWLEDGMENT

We would like to thank our employer, ZHAW, for allowing us to carry out a project that we are very passionate about. Furthermore, we would like to extend our gratitude to our coworkers that proofread this paper and significantly contributed to the quality of this work.

References

[1] "What is memory safety and why does it matter?," ISRG, [Online]. Available: https://www.memorysafety.org/docs/memory-safety/. [Accessed 06. May 2022].

[2] "A Brief History of Malware," Lastline, [Online]. Available: https://www.lastline.com/blog/history-of-malware-its-evolution-and-impact/. [Accessed 06. May 2022].

[3] "2021 in Memory Unsafety - Apple's Operating Systems," Fish in a Barrel, [Online]. Available: https://langui.sh/2021/12/13/apple-memory-safety/. [Accessed 06. May 2022].

[4] "We need a safer systems programming language," Microsoft Security Response Center, [Online]. Available: https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/. [Accessed 06. May 2022].

[5] "Queue the Hardening Enhancements," Google Security Blog, [Online]. Available: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html. [Accessed 07. May 2022].

[6] "A Year in Review of 0-days Used In-the-Wild in 2021," Project Zero, [Online]. Available: https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html. [Accessed 13. May 2022].

[7] "Laying the foundation for Rust's future," Rust Blog, [Online]. Available: https://blog.rust-lang.org/2020/08/18/laying-the-foundation-for-rusts-future.html. [Accessed 07. May 2022].

[8] "Announcing the Rust Foundation to the World," Rust Foundation, [Online]. Available: https://foundation.rust-lang.org/news/2021-02-08-hello-world/. [Accessed 07. May 2022].

[9] S. Klabnik, *How Rust Views Tradeoffs,* London: QCon, 2019.

[10] "rustup.rs - The Rust toolchain installer," Rust Foundation, [Online]. Available: https://rustup.rs/. [Accessed 12. May 2022].

[11] "The Cargo Book," Rust Foundation, [Online]. Available: https://doc.rust-lang.org/stable/cargo/. [Accessed 12. May 2022].

[12] "doctest - Test interactive Python examples," [Online]. Available: https://docs.python.org/3/library/doctest.html. [Accessed 13. May 2022].

[13] "doctest: Test interactive Haskell examples," [Online]. Available: https://hackage.haskell.org/package/doctest. [Accessed 13. May 2022].

[14] "Characteristics of Object-Oriented Languages," The Rust Programming Language, [Online]. Available: https://doc.rust-lang.org/book/ch17-01-what-is-oo.html. [Accessed 08. May 2022].

[15] "Inheritance as a Type System and as Code Sharing," The Rust Programming Language, [Online]. Available: https://doc.rust-lang.org/book/ch17-01-what-is-oo.html#inheritance-as-a-type-system-and-as-code-sharing. [Accessed 08. May 2022].

[16] "Open Source at Ferrous Systems," Ferrous Systems, [Online]. Available: https://ferrous-systems.com/open-source/. [Accessed 08. May 2022].

[17] "Ferrocene: Rust for Critical Systems," Ferrous Systems, [Online]. Available: https://ferrous-systems.com/ferrocene/. [Accessed 08. May 2022].

[18] "ISO 26262," Solid Sands, [Online]. Available: https://solidsands.com/safety/iso-26262. [Accessed 08. May 2022].

[19] P. Anderson, "The Use and Limitations of Static-Analysis," CrossTalk - Journal of Defense Software Engineering, vol. 21, 2008.

[20] "Zig," Zig, [Online]. Available: https://ziglang.org/. [Accessed 07. May 2022].

[21] J. Brandon, "How safe is zig?," [Online]. Available: https://www.scattered-thoughts.net/writing/how-safe-is-zig/. [Accessed 07. May 2022].

[22] "Internet Security Research Group," ISRG, [Online]. Available: https://www.abetterinternet.org/. [Accessed 07. May 2022].

[23] "Project Prossimo," ISRG, [Online]. Available: https://www.memorysafety.org/. [Accessed 07. May 2022].

[24] "RFC for Rust in the Linux Kernel," Linux Kernel Mailing List, [Online]. Available: https://lkml.org/lkml/2021/4/14/1023. [Accessed 12. May 2022].

[25] "Linus Torvalds on C++," cat -v, [Online]. Available: http://harmful.cat-v.org/software/c++/linus. [Accessed 12. May 2022].

[26] "The Embedded Rust Book," The Embedded Rust Book, [Online]. Available: https://docs.rust-embedded.org/book/intro/index.html. [Accessed 10. May 2022].

[27] "embedded-hal," Github, [Online]. Available: https://github.com/rust-embedded/embedded-hal. [Accessed 10. May 2022].

[28] "Design Contracts," The Embedded Rust Book, [Online]. Available: https://docs.rust-embedded.org/book/static-guarantees/design-contracts.html. [Accessed 10. May 2022].

[29] "CVE-2020-6007 Detail," The MITRE Corporation, [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2020-6007. [Accessed 12. May 2022].

[30] "nRF52840," Nordic Semiconductor, [Online]. Available: https://www.nordicsemi.com/Products/nRF52840. [Accessed 11. May 2022].

[31] "Zephyr Project," The Linux Foundation, [Online]. Available: https://www.zephyrproject.org/. [Accessed 12. May 2022].

[32] "OpenThread," OpenThread, [Online]. Available: https://openthread.io/. [Accessed 11. May 2022].

[33] W. Zhou, L. Guan, P. Liu and Y. Zhang, "Good Motive but Bad Design: Why ARM MPU Has Become an Outcast in Embedded Systems," August 2019. [Online]. Available: https://arxiv.org/pdf/1908.03638.pdf. [Accessed 10. May 2022].

[34] "MbedTLS," Linaro Limited, [Online]. Available: https://www.trustedfirmware.org/projects/mbed-tls/. [Accessed 12. May 2022].

[35] "RustCrypto," Github, [Online]. Available: https://github.com/RustCrypto. [Accessed 12. May 2022].

[36] "The Computer Language Benchmarks Game," Debian, [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html . [Accessed 12. May 2022].