Rust for Embedded Systems: Current State and Open Problems

Ayushi Sharma*
Purdue University
West Lafayette, USA
sharm616@purdue.edu

Shashank Sharma*
Purdue University
West Lafayette, USA
sharm611@purdue.edu

Sai Ritvik Tanksalkar
Purdue University
West Lafayette, USA
stanksal@purdue.edu

Santiago Torres-Arias
Purdue University
West Lafayette, USA
torresar@purdue.edu

Aravind Machiry
Purdue University
West Lafayette, USA
amachiry@purdue.edu

Abstract

Embedded software is used in safety-critical systems such as medical devices and autonomous vehicles, where software defects, including security vulnerabilities, have severe consequences. Most embedded codebases are developed in unsafe languages, specifically C/C++, and are riddled with memory safety vulnerabilities. To prevent such vulnerabilities, Rust, a performant memory-safe systems language, provides an optimal choice for developing embedded software. Rust interoperability enables developing Rust applications on top of existing C codebases. Despite this, even the most resourceful organizations continue to develop embedded software in C/C++.

This paper performs the first systematic study to holistically understand the current state and challenges of using Rust for embedded systems. Our study is organized across three research questions. We collected a dataset of 6,408 Rust embedded software spanning various categories and 6 Static Application Security Testing (SAST) tools. We performed a systematic analysis of our dataset and surveys with 225 developers to investigate our research questions. We found that existing Rust software support is inadequate, SAST tools cannot handle certain features of Rust embedded software, resulting in failures, and the prevalence of advanced types in existing Rust software makes it challenging to engineer interoperable code. In addition, we found various challenges faced by developers in using Rust for embedded systems development.

CCS Concepts

• **Security and privacy → Embedded systems security**; • **Computer systems organization → Embedded software**; *Real-time operating systems*; • **Hardware** → Safety critical systems.

Keywords

Rust, Deep Embedded Systems, Security

*Both authors contributed equally to this research.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690275

ACM Reference Format:
Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. 2024. Rust for Embedded Systems: Current State and Open Problems . In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690275

1 Introduction

Our dependence on embedded devices (*e.g.,* IoT devices), has significantly increased, controlling various aspects of our daily lives, including homes [11], transportation [9], traffic management [130], and the distribution of vital resources like food [102] and power [97]. The adoption of these devices has seen rapid and extensive growth, with an estimated count of over 50 billion devices [8]. Vulnerabilities in these devices have far-reaching consequences [4, 14] due to the pervasive and interconnected nature of these devices, as exemplified by the infamous Mirai botnet [83].

Most embedded software are developed in "unsafe" (i.e., not memory-safe) languages, specifically C/C++, because of the low memory footprint, good performance, and the availability of extensive support software. It is well-known that software developed in unsafe languages is prone to security vulnerabilities, especially memory safety vulnerabilities [25, 89, 135]. Likewise, embedded systems are riddled with security vulnerabilities [7, 13, 62, 136]. The most recent URGENT/11 [15] vulnerabilities in VxWorks that affected millions of medical [56], SCADA systems [142], and industrial controllers [22] are all because of memory safety (spatial) violations. It is important to ensure that embedded systems do not contain memory-safety issues. Automated memory-safety retrofitting techniques [32, 48, 64, 92, 93, 131] based on compile-time instrumentation add significant overhead (both space and runtime) and are inapplicable to resource-constrained embedded systems.

Our analysis (details in Our Extended Report [126]) of security vulnerabilities in various Real Time Operating Systems (RTOSes) (an important class of embedded software) for the past ten years shows that 59 (54.2%) of them are memory corruption vulnerabilities, *i.e.,* spatial or temporal memory issues. It is important to use memory-safe languages to prevent such vulnerabilities. Furthermore, recently, the White House released a report [140] requiring future software to be developed in memory-safe languages. Traditional memory-safe languages, such as Java, have high overhead and are not suitable for embedded systems. Rust [121] is a memory-safe language that is shown to have comparable performance as native code. Furthermore, Rust can easily interoperate with existing unsafe codebases [61], enabling incremental adoption. Rust team has a special focus on embedded systems [112], and several works [74, 75] demonstrate the feasibility of engineering a complete embedded software stack in Rust. Furthermore, Rust popularity is rising [101], and it is now adopted in Linux kernel [79] and Android [108]. Unfortunately, embedded systems are continuing to be developed in C. Even the most resourceful organizations, such as Microsoft, continue to develop embedded systems in C, as demonstrated by their recent Azure RTOS [18]. Previous works [55, 145] investigated the challenges of adopting Rust for regular software. However, no work tries to understand factors affecting the use of Rust for embedded systems development.

In this paper, we perform the first systematic study to holistically understand the issues in using Rust for developing embedded systems. Specifically, we explored the following research questions:

- **RQ1: Software Support.** How effective (quantity and quality) is the existing Rust software support for embedded system development?
- **RQ2: Interoperability of** Rust**.** Given that most of the existing embedded systems are in C, how well can Rust interoperate with existing C codebases? and what are the challenges specific to embedded codebases?
- **RQ3: Developers Perspective.** What challenges do developers face in using Rust for embedded system development?

We collected a dataset of 6,408 Rust embedded software packages (or crates) spanning various categories and 6 SAST tools. We performed a systematic analysis of our dataset and surveys with 225 developers to investigate our research questions. Our study revealed several interesting findings (16), drawbacks of existing tools on embedded crates, and open problems (8) to increase the adoption of Rust for embedded systems. A few interesting findings include the following: Embedded crates use more (~2X) `unsafe` blocks than non-embedded crates, significantly increasing the possibility of memory safety issues. However, existing techniques to isolate `unsafe` blocks are not applicable to embedded systems. Existing developer support tools related to Rust, such as c2rust, fail on majority of embedded codebases, as these tools fail to handle the diverse build systems and execution semantics of embedded systems. The state-of-the-art Rust SAST tools perform poorly on embedded crates. The superior type-system of Rust makes it challenging to engineer interoperable embedded systems code. Our observations are in line with the developer survey, and many developers consider the Rust documentation for embedded systems poorly organized and want the documentation to contain more examples. In summary, the following are our contributions:

- **Software Study:** We perform a systematic study of the Rust software ecosystem to support the use of Rust for embedded applications and highlight opportunity areas for adoption.
- **Tool Study:** We systematically studied the effectiveness of various (9) Rust related tools, *i.e.,* SAST tools, quality checking, and conversion tools, on embedded crates and identified various weaknesses specific to embedded systems.
- **Developer Aspects:** We performed a large-scale developer survey (with 225 developers) that highlights the challenges for slow adoption of Rust for embedded applications.
- **Dataset, Findings, and Open Problems** [1]**:** We curated a set of 6,408 embedded Rust crates cataloged into various categories along with the necessary infrastructure to run analysis tools. Our findings shed light on challenges in adopting Rust for embedded systems, insights into open problems, and possible research directions.

2 Background

This section provides the necessary background information for the rest of our work.

2.1 Embedded Systems

Embedded systems are designed to perform a designated set of tasks in a resource-constrained environment and on battery-powered devices. There are several ways to categorize embedded systems. Previous work [91] categorizes embedded systems based on underlying Operating System (OS).

(a) Type-1 systems have feature-rich general-purpose OSes retrofitted for embedded systems.

(b) Type-2 systems or constrained devices [26] use specialized embedded OSes, which are usually designed as Real Time Operating System (RTOS), *e.g.,* WEMO Light controller [138] running FreeRTOS [52].

(c) Type-3 systems do not use OS abstractions and are rarely used in commercial products.

Previous work [11] shows consumer IoT devices, such as door knobs and temperature controllers, are mostly Type-2, which we primarily focus on. Type-2 systems execute on battery-powered and resource-constrained Microcontroller Units (MCUs). These systems have a lot of diversity in terms of hardware (MCU and peripherals) and supported software [91, 139]. For instance, there are 31 different RTOSes [98]. To handle this diversity, Type-2 systems have a layered design [128] (illustrated in our Extended Report [126]). Application logic is implemented in tasks managed by an RTOS.
Execution Semantics. The application and all the layers are compiled into a single monolithic binary and flashed onto the on-chip flash memory. On reset, the contents of the memory are loaded into RAM, and execution starts from a pre-defined address, *i.e.,* start or reset address. The tasks get scheduled per the scheduling policy, and handlers get triggered on corresponding events.

2.2 Rust

Rust is a programming language created by Mozilla to build efficient and safe low-level software [27, 68, 71, 121, 124]. Rust is targeted to achieve performance comparable to programs written in C while avoiding many safety issues in C, including concurrency and memory safety bugs. This section provides a brief overview of Rust's safety features. We recommend the Rust's official book [121] for a comprehensive understanding of these features.

[1]https://zenodo.org/records/12775715

Features and Safety Guarantees. Rust has several features, such as scopes, borrowing rules [120], single ownership [118], and lifetimes [115], which force developers to follow certain practices enabling verification of memory safety properties (mostly) at compile time. For instance, all read-write variables should be explicitly marked as mutable (*i.e.,* `mut`). Rust provides both spatial and temporal memory safety. We provide a discussion of these guarantees in Our Extended Report [126]

Unsafe Rust. Rust features can be too restrictive in a few cases. For instance, Rust requires all global variables to be read-only, *i.e.,* disallows `mut`. Similarly, we may need to call a C library function, which is also not allowed. Rust provides `unsafe` blocks [119] to relax these restrictions and enable interaction with external language (or foreign) functions. Arbitrary regions of code can be enclosed in an `unsafe` directive, and such code will be permitted certain (otherwise disallowed) actions, such as modifying a mutable global variable, dereferencing a raw pointer, calling an unsafe or external method, etc.

Foreign Function Interface (FFI) Support. Rust supports easy interaction with functions written in foreign languages through its Foreign Function Interface (FFI) [61]. Specifically, such functions need to be annotated with special attributes, which enables the Rust compiler to generate appropriate code respecting the ABI of the target language.

Build System and Package Management. Rust uses an integrated and easy-to-use build system and package manager called **Cargo** [110], which downloads library packages, called crates, as needed, during builds. Developers specify the build configuration along with all dependencies in a `.toml` file [133] — an organized key-value text file. Rust has an official community package registry called crates.io [36], which (as of 29 April 2024) has more than 144K crates (*i.e.,* libraries) – a 200% increase over the last two years.

Rust **Compilation Attributes**. Rust supports attributes or configurations that enable compilation specialization. These attributes can be at various levels, *e.g.,* crate level, file level, function level, etc. **`no_std` attribute [46]** is a crate-level attribute that avoids linking the entire standard module and results in small binaries. Embedded software in Rust should use this attribute to produce a self-contained binary independent of OS abstractions. A `no_std` compatible crate should also have all its dependencies to be `no_std` compatible too.

3 Study Methodology

Our study aims to perform a holistic analysis to understand various aspects regarding usage of Rust for embedded systems. We aim to answer the following research questions:

- **RQ1: Software Support (§4)**: How good is the software support for developing Rust based embedded systems?
- **RQ2: Interoperability (§5)**: How effective is the interoperability support of Rust to deal with existing C based embedded codebases?
- **RQ3: Developers Perspective (§6)**: What is developers' perspective on using Rust for embedded systems?

3.1 Embedded Software Dataset

Our goal is to collect Rust crates that are applicable to embedded systems, *i.e.,* `no_std` compatible, and can be built using one of the embedded toolchains. We also want to identify the necessary compilation steps for all the collected crates.

3.1.1 Crates collection. As mentioned in §2.2, crates.io is the official repository for all Rust crates (*i.e.,* libraries). However, there are other well-known sources, such as Rust-embedded project [113] and arewertosyet.com, that also contain embedded Rust projects. We used a two-pronged approach to collect our embedded Rust dataset.

- **Crawling crates.io:** We crawled crates.io (in Feb 2024) and got all the crates that are `no_std` compatible. This is not trivial as crates can declare `no_std` compatibility at various levels. For instance, `arduino_nano_connect v0.6.0` [34] crate declares `no_std` compatibility at the crate level (*i.e.,* in `lib.rs` file). In contrast, `futures-executor v0.3.30` [39] crate uses `cfg` attribute to have only selected code blocks compile for `no_std` environment. We perform lightweight static analysis to identify all such crates.
- **Well-known Sources**: We collected additional crates by crawling other well-known sources, specifically Rust-embedded project and arewertosyet.com.

After deduplication, we collected 11,002 unique crates.

3.1.2 Identifying Stable Crates. We tried to build crates using a stable version of Rust and the corresponding compiler. However, we identified that 2,025 (18.4%) embedded crates depend on unstable Rust versions, specifically nightly versions [2]. These versions contain unstable Rust features and might pose threats to the security guarantees of Rust. This is also reflected in one of the concerns (in §6.3) raised by developers in using Rust for embedded systems. We only considered those that build on the stable version of Rust, specifically 1.77.2. This resulted in 8,977 crates.

3.1.3 Compilation Validation. The `no_std` compatibility alone is a necessary condition but *not sufficient* for a crate to be usable on embedded systems. For instance, `oc-wasm-futures` [41] crate is `no_std` compatible but is for WebAssembly target, which is not an embedded architecture.

In this step, our goal is to validate crates to check for their applicability to embedded targets and identify the corresponding build commands.

Identifying Build Command: All crates can be built using `cargo build`, which uses the default configuration specified in the crate's `cargo.toml` file. However, not all crates have their default configuration to be `no_std`, *i.e.,* the default build step (`cargo build`) may not build `no_std` compatible version. Such crates require special configuration flags to be passed to the build command, *e.g.,* we need to use

`cargo build --no-default-features --features no_std` to build `no_std` variant of `async_cell`. Developers specify such flags through Rust's conditional compilation attributes [1] (`cfg_attr`) as a propositional logic formula.

For instance, `#![cfg_attr(all(feature = "no_std", not(feature = "std")), no_std)]` (in `resize v0.8.4` crate) indicates that we need to pass `no_std` flag and not pass `std` to build for `no_std`.

We use a lightweight static analysis technique to identify the appropriate build command. First, we identify all `cfg_attr` directly corresponding to `no_std` (*i.e.,* containing `#![cfg_attr(..., no_std)]`) Second, we analyze the propositional formula to identify the flags that must be enabled or disabled for `no_std`. Our technique was able to find the build commands for 8,148 (90.77%) of crates.

The rest 9.23% failed because of the following reasons: (i) *Incorrect attributes:* Here, crates have an incorrect `cfg_attr` specification. For instance, `zero-crypto v0.1.11` crate has wrong flag name `(featusre = "std")` (Correct: (feature = "std")); (ii) *Incorrect dependencies:* As mentioned in §2, for a crate to be `no_std` compatible all its dependencies should also be `no_std`. However, few crates use dependencies that are either not `no_std` compatible or incorrectly configured. For instance, `linux-kvm v0.2.0` crate depends on `linux-io v0.6.0` crate, which is not `no_std` compatible.; (iii) *Complex attributes specification:* In our analysis, we consider only directly related flags, *i.e.,* those specified along with `no-std` in `cfg_attr`. However, there could be conditional compilation flags that are indirectly related. For instance, `ab_glyph v0.2.23` crate requires `--features="libm"` flag. As we explain in Our Extended Report [126], this flag dependency is specified indirectly and accurately identifying such flags is a combinatorial problem [6].

Open Problem P0.1: We need techniques to automatically identify embedded system specific (*i.e.,*`no_std` compatible) build configurations for Rust crates — this also enables identifying mistakes in build configurations (a prevalent problem). One possible approach is to map the dependencies into a boolean formula for constraint solver and use the solution to derive the appropriate flags.

3.1.4 Embedded Targets Filtering. There are 23 embedded targets (85 total targets) supported by the latest stable version of `rustc` (version: 1.77.2). For all the crates for which we identified the build commands, we further filtered out crates that did not build for any of the embedded targets. For instance, the `no_std` variant of `winapi v0.3.9` crate is excluded because it requires an underlying operating system environment, which is not present in embedded targets. This resulted in a total of 6,408 crates after filtering out 2,569 crates. Although our study focuses on type-2 systems, *our crates are not exclusively type-2.* For instance, type-1 systems exist for `aarch64-cpu`, one of our targets.

3.1.5 Categorization. Based on the functionality, we categorize each embedded crate into eight categories (Tbl. 1). We will present details of these categories in §4.1. We created a Multi-class Random Forest (MCRF) classifier [28] to categorize a given crate. We manually categorized 2000 crates into various categories. Using this as ground truth, we created an MCRF classifier with an F1-score of 82%. We used our MCRF classifier to categorize the rest of the crates.

Summary. We collected a total of 6,408 embedded crates along with appropriate build commands that produce `no_std` compatible binary.

4 RQ1: Software Support

In this research question, we want to assess the existing software support for engineering embedded systems in Rust. We plan to investigate the categories of support software that aid in common software engineering activities. Specifically:

- **(For Development) Libraries and Support Software:** As explained in §2, applications in embedded systems are developed atop an RTOS and need necessary libraries that enable communicating with peripherals and provide certain common functionality (*e.g.,* network protocols).
- **(For Testing) SAST Tools:** These are an integral part of software development [107]. We need to have effective SAST tools to ensure the quality of newly developed Rust based embedded systems.
- **(To Handle Existing Codebases) C to Rust Conversion Tools:** Given that most existing embedded codebases are in C, we should have tools to convert C to Rust effectively.

4.1 Libraries and Support Software

The available software support, *i.e.,* crates, can be broadly categorized into *hardware support crates* and *utility crates.*

Table 1: Categorization of all available embedded crates.

Category Abbr	Type	Crates	
		Total	Wrapper Crates (%Total)
Rtos	RTOS Crates	6	2 (33.33%)
Dr	Driver Crates	466	31 (6.65%)
Hal	HAL Crates	57	29 (50.88%)
Bsp	Board Support Package	114	100 (87.72%)
Pac	Peripheral Access Crates	565	439 (77.70%)
Arch	Architecture Support Crates	15	9 (60.00%)
Util	Utility Crates	4,764	173 (3.63%)
Uncat	Others	421	30 (7.18%)
	TOTAL	6,408	813 (12.69%)

4.1.1 Hardware Support Crates. These provide software abstraction to interact with the hardware *i.e.,* MCU, Peripherals, etc. There are at least 43 different MCU families with various peripherals [5, 90, 132]. We use the following categories to further categorize based on the type of interactions the crates provide. Tbl. 1 shows the summary of different categories of crates available for embedded systems development.

- **Architecture Support:** These help in interacting with the processor and are Instruction Set Architecture (ISA) dependent. For instance, the `aarch64-cpu` crate [111] provides the function `SPSR_EL2.write` to write to the Saved Program Status Register (SPSR) at EL2 exception level on aarch64 processors. *For embedded processors (i.e., Reduced Instruction Set Computer (RISC) ISAs), there are support crates for ARM, MIPS, and RISC-V.*
- **Peripheral Access:** These provide necessary functions to access peripherals on different MCUs. *Out of 43 different MCU families, peripheral crates are currently available for only 16 (37%).* Most of these crates are generated using `svd2rust` utility [42], which automatically converts CMSIS-SVD [63] file (XML description of ARM Cortex-M processors) into Rust structs. Consequently,

most of these crates are for ARM Cortex-M family processors. However, other MCUs, such as AVR-based ATMEGA48PB, do not have SVD files but rather have `.atdf` files. There exist crates such as `atdf2svd` [35] to convert these into CMSIS-SVD format, but these tools are not robust and have issues.

Open Problem P1.1: We need effective techniques to automatically generate peripheral access crates for non-ARM architectures. Recent advancements in LLM-assisted techniques [143] show promise in solving this problem.

- **HAL Implementation (HAL):** These are implementations of `embedded-hal` [38], a common Hardware Abstraction Interface for various MCUs. These provide higher-level functions than peripheral crates, which just provide structures encapsulating peripheral registers. For example, `GPIO::write` is a function provided by `embedded-hal`, which involves multiple interactions through GPIO registers. *The HAL crates are available for 14 (32%) MCU families.* Unlike peripheral access crates, HAL crates are not automatically generated but are manually engineered. Consequently, a lot of variance exists in MCU families having HAL crates. For instance, Espressif MCUs (with Xtensa ISA) has HAL crates [114] but does not have peripheral crates.
- **Board Support (BSP):** These crates help in bootstrapping an MCU for an RTOS. Specifically, these included bootloaders and other code to initialize and recognize other hardware peripherals. BSPs are built using HAL and peripheral traits and expose higher-level functions to operate the underlying MCU or System on Chip (SoC). For instance, `hifive1` BSP crate [123] (for HiFive1 boards) exposes a function `configure_spi_flash` which uses `e310x_hal` [122] HAL crate to configure SPI Flash with maximum speed. *There are BSP crates for 19 (44%) different boards.* Unlike peripheral or HAL crates, BSP crates are specific to each board —- a combination of MCU and peripherals.

Open Problem P1.2: Recent work [128] exposes layering violations in C embedded systems, *i.e.,* components breaking the layered abstraction, *e.g.,* HAL crate not using peripheral crates. However, no such work exists for Rust crates.

- **Driver Crates:** These are device drivers and expose functions to access various aspects of a device. For instance, `eeprom24x` driver crate [37] provides the necessary functions (*e.g.,* `read_byte`) to access 24x series serial EEPROMs.
- **RTOS Crates:** These are complete RTOSes, which expose necessary functions for task creation and synchronization, thus enabling easy creation of embedded applications.

Finding RQ1.1: Existing hardware support crates mainly target ARM Cortex-M family MCUs and boards. Although, there are ongoing efforts [109] to improve support for other family MCUs such as AVR. It is still a work in progress, and more efforts are required.

4.1.2 *Utility Crates.*

These are hardware-independent embedded crates (*i.e.,* `no_std`) that provide various capabilities for embedded systems development. For instance, `tinybmp` embedded crate [43] provides functions to parse BMP images.

Despite the existence of a large number of utility crates in the Rust ecosystem, only 4,764 can be used in embedded systems because of the requirement to be independent of OS abstractions, *i.e.,* should not use Rust's `std` crate (or be `no_std` compatible). However, it is not easy to convert a crate to be `no_std` [104] compatible as it requires the ability to perform semantic refactoring of the crate and its dependencies. Our Extended Report has an example.

Open Problem P1.3: We need techniques to automatically convert Rust crates to be `no_std` compatible to enable existing large quantity of crates to be usable in embedded systems. Recent work by Sharma *et al.,* [127] demonstrates a possible approach using type-based conditional compilation.

4.1.3 *Quality of Embedded* Rust.

At a high level, as shown by the last column of Tbl. 1, many (813 (12.69%)) of the crates are just wrappers around C libraries (details in Our Extended Report [126]). We also consider crates that depend on a wrapper crate to be wrapper crates. These crates are susceptible to the bugs in corresponding C libraries. In other words, vulnerabilities in the wrapped libraries can be exploited to get complete control of the corresponding Rust program. This problem has received considerable attention, and several works try to isolate code running as part of libraries (or in general `unsafe` blocks) from the rest of the crate. These techniques depend on special hardware features [20, 58, 67, 80, 106], specifically Intel's Memory Protection Key (MPK) or OS abstractions, such as `mprotect` [10], IPC mechanism [99], sandboxing [72] — making them inapplicable to type-2 embedded systems, *i.e.,* RTOS based embedded systems that run on MCUs.

Open Problem P1.4: We need techniques (applicable also for embedded systems) to isolate Rust code from `unsafe` code, *i.e.,* techniques that do not depend on hardware features, OS abstractions, and have low overhead. Recent techniques [65, 66] on C-based embedded software compartmentalization demonstrate possible approaches. However, these should be customized for Rust.

Code Quality: We use the following tools to further assess embedded crates' code quality.

- Qrates **[17]:** This tool finds instances of various `unsafe` idioms, *i.e.,* blocks, functions, traits, and trait implementations. Unfortunately, the tool failed on 405 (6.30%) crates. We provide a categorization of failures in Our Extended Report [126]. Out of the remaining 6003 crates, 2634 (43.88%) contain at least one `unsafe` idiom. This is much higher than in non-embedded crates where only 23.6% crates (as reported in [17]) contain `unsafe` idioms. Tbl. 2 shows the results along with top three reasons for `unsafe`ness. Note that the percentages are not cumulative, *i.e.,* there could be multiple reasons for an `unsafe` block. These reasons differ from regular crates, indicating the need for different design decisions when creating analysis tools for embedded crates.
- cla-metrics **[86]:** Recently, Mergendahl *et al.,* [86] demonstrated the feasibility of Cross-Language-Attacks, wherein interactions of Rust with C/C++ could result in security vulnerabilities. They released cla-metrics, a tool to identify these cross-language interaction points. cla-metrics works on binaries and require ELF files with specific sections. As we showed in Tbl. 1, there are 813 wrapper crates, which means these contain at least one call from Rust to C/C++, *i.e.,* a transfer point. Interestingly, cla-metrics found only 198 crates with interaction points. These results indicate potential issues with the cla-metrics tool and

Table 2: Summary of Qrates results on embedded crates (uf: Call to Unsafe Function, ptr: Derefercing raw pointer, mstat: Use of Mutable Static, estat: Use of Extern Static, asm: Use of Inline Assembly, union: Access to Union Field)

Category	Num. Crates Successful (% of Total from Tbl. 1)	Number of Crates (% of Successful) having					Top 3 Reasons for Unsafe Usage
		Unsafe Blocks	Unsafe Functions	Unsafe Trait Impl	Unsafe Trait	At least one unsafe idiom	
Rtos	3 (50.00%)	3 (100.00%)	2 (66.67%)	2 (66.67%)	0 (0.00%)	3 (100.00%)	ud (82%), mstat (29%), estat (6%)
Dr	423 (90.77%)	70 (16.55%)	55 (13.00%)	11 (2.60%)	3 (0.71%)	91 (21.51%)	ud (82%), ptr (19%), mstat (2%)
Hal	42 (73.68%)	28 (66.67%)	20 (47.62%)	12 (28.57%)	9 (21.43%)	31 (73.81%)	uf (72%), ptr (38%), mstat (2%)
Bsp	89 (78.07%)	25 (28.09%)	23 (25.84%)	4 (4.49%)	0 (0.00%)	29 (32.58%)	uf (88%), ptr (77%), mstat (2%)
Pac	560 (99.12%)	508 (90.71%)	538 (96.07%)	528 (94.29%)	5 (0.89%)	547 (97.68%)	uf (91%), ptr (18%), mstat (1%)
Arch	10 (66.67%)	9 (90.00%)	10 (100.00%)	1 (10.00%)	1 (10.00%)	10 (100.00%)	uf (52%), asm (31%), ptr (22%)
Util	4,473 (93.89%)	1,591 (35.57%)	1053 (23.54%)	554 (12.39%)	210 (4.69%)	1,790 (40.02%)	uf (89%), ptr (16%), union (1%)
Uncat	403 (95.72%)	102 (25.31%)	78 (19.35%)	30 (7.44%)	8 (1.99%)	133 (33.00%)	uf (84%), ptr (25%), mstat (2%)
Total	6003 (93.68%)	2336 (38.91%)	1779 (29.64%)	1143 (19.04%)	236 (3.93%)	2634 (43.88%)	uf (90%), ptr (18%), mstat (1%)

Table 3: cla-metrics results.

Category	Num. Crates Successful (% of Total from Tbl. 1)	Num. Crates having at least one Transfer Pt.
Rtos	6 (100.00%)	0 (0.00%)
Dr	462 (99.14%)	0 (0.00%)
Hal	56 (98.25%)	0 (0.00%)
Bsp	113 (99.12%)	0 (0.00%)
Pac	562 (99.47%)	0 (0.00%)
Arch	15 (100.00%)	0 (0.00%)
Util	4727 (99.22%)	58 (1.23%)
Uncat	416 (98.81%)	14 (3.37%)
Total	6,357 (99.20%)	73 (1.13%)

we found that an important class of such transfer points that cla-metrics misses is indirect function calls. Indirect function calls are common in embedded systems due to their event driven nature. Recent works [30] show that employing CFI mechanism through LLVM can help detect indirect calls. Although as we see in 4.3.3, embedded systems fail to build with clang. We need more work in the area. cla-metrics uses the differences in name mangling used by Rust and C++ to determine such transfer points. This information would not be available for indirect function calls and hence cla-metrics misses out these.

Security Implications: The prevalence of `unsafe` idioms indicates that developers should be more cautious in using embedded crates. Moreover, the robustness issues in analysis tools indicate that security researchers should consider embedded crates as part of their evaluation.

Finding RQ1.2: Compared to non-embedded crates, many embedded crates (48.5% v/s 23.6%) contain `unsafe` Rust code idioms.
Finding RQ1.3: cla-metrics fails to identify cross-language interactions through indirect calls *e.g.*, calls through function pointers.

4.2 SAST Tools

As we show in §4.1.3, embedded crates contain a large amount of `unsafe` blocks. The presence of `unsafe` blocks potentially violates Rust's safety guarantees and results in various memory safety issues [17]. It is important to use SAST tools for embedded systems development in Rust. We investigate the effectiveness of state-of-the art Rust SAST tools on embedded crates.

Table 4: Failure reasons of SAST tools and the number of affected crates. Our Extended Report [126] contains detailed and examples of failures.

Failure Reason	Affected Tools and Crates	Total
Toolchain Incompatibility	FFIChecker (2559, 39.93%) Rudra (2547, 39.75%) Yuga (539, 8.41%) SafeDrop (166,2.59%) rCanary (156, 2.43%) Lockbud (30, 0.46%)	2,692
Tool Crashes	FFIChecker (67, 1.046%) rCanary (9,0.14%) SafeDrop (1, 0.015%) Lockbud (5, 0.78) Yuga (1, 0.02%)	89
No binary target support	SafeDrop (27, 0.42%) rCanary (25, 0.39%) FFIChecker (6, 0.094%) Rudra (4, 0.04%)	27
Ignoring Project-Specific Configurations	Yuga (19, 0.30%) Rudra (6, 0.09%) FFIChecker (2, 0.03%) SafeDrop (2, 0.03%) rCanary (1, 0.02%)	21
Timeouts (large crates)	rCanary (16, 0.25%)	16
Rustc version incompatibility	FFIChecker (8, 0.12%)	8
Unknown Reasons	Yuga (7, 0.11%)	7

4.2.1 Tool Selection. The recent study by Ami *et al.*, [12] shows that developers are more likely to use SAST tools that do not require any configuration and can be directly used on a software project. Following this, we aim to collect state-of-the-art and readily usable SAST tools. Specifically, these tools should run directly on a given crate and not require any configuration. We searched Rust

Table 5: Summary of Rust SAST tools evaluated as part of the study and the features that are supported (✓) or not (✗), type of reports (insufficient information , missing relevant details , or detailed report), along with references to results.

Tool Name	Target Bug Types	Techniques Used	Flow-Tracking: Require	Flow-Tracking: Across Unsafe blocks	Flow-Tracking: Across FFI Boundaries	Flow-Tracking: Handles Async/ Indirect Flows	Report Type	Result Reference
Lockbud [103]	Concurrency, Memory Safety	CallGraph Analysis, Points-to Analysis, Dataflow Analysis	✓	✗	✗	✗	Source Level Traces (Our Extended Report [126])	Refer Our Extended Report [126]
Rudra [19]	Temporal & Spatial safety	Taint Analysis, Dataflow analysis	✓	✓	✗	✗	Source Level Traces (Our Extended Report [126])	
Yuga [94]	Temporal Safety	Taint Analysis, Alias Analysis	✓	✓	✗	✗	Detailed Source Level Traces (Our Extended Report [126])	
rCanary [45]	Temporal Safety	Dataflow Analysis, Constraint solving	✓	✓	✗	✗	sat/unsat (Our Extended Report [126])	
FFIChecker [77]	Temporal Safety	Taint Analysis, Alias Analysis	✓	✓	✓	✗	Generic warning (Our Extended Report [126])	
SafeDrop [44]	Temporal Safety	Dataflow Analysis, Alias analysis	✓	✓	✗	✗	Affected Function name (Our Extended Report [126])	

forums and the past five years' proceedings of top-tier security and software engineering conferences and collected the set of readily usable SAST tools. We filtered out tools that did not satisfy our requirements. For instance, we did not select MirChecker [76] because it requires configuring the abstract domain and specifying analysis entry points. After filtering such tools, our investigation resulted in six tools as summarized in Tbl. 5.

Almost all tools except for Lockbud focus on identifying temporal safety issues, *e.g.,* incorrect lifetimes, and multiple drops. All these tools are based on flow-tracking as indicated by ✓ under the *Require* column.

4.2.2 Qualitative Assessment. As presented in §4.1.3, embedded Rust crates have a higher percentage of unsafe blocks, use FFI functions (*i.e.,* interact with C libraries), and use indirect (or function pointer) calls. SAST tools should be able to handle these idioms to be effective on embedded crates.

Supported Features: We referred to the research papers on the corresponding tools and created simple examples to identify their capabilities to handle idioms common in embedded crates. The various columns under *Flow-Tracking* indicate whether each of these idioms is either supported (✓) or not (✗) by the corresponding tools. All tools, except for Lockbud, handle flows across `unsafe` blocks. None of the tools handle data-flows through indirect calls (*i.e.,* function pointer calls) — which is one of the common idioms in embedded systems (§2.1 and [128]). Except for FFIChecker, none of the tools handle flows across FFI boundaries, another common usage in embedded crates.

Usability: Despite the existence of standard formats, such as SARIF [3], Rust SAST tools employ ad-hoc ways to report their warnings. As shown in the last column of Tbl. 5, these reports do not always contain the necessary information to triage the underlying defect. The Tbl. 5 also contains references to the examples of corresponding warnings. *All tools, except for* Yuga, *report their findings in an ad-hoc and hard-to-analyze manner.* rCanary and FFIChecker just provide a single-line warning without any details about the source location — which makes these warnings almost impossible to analyze. Lockbud, Rudra, and SafeDrop provide source level traces. However, the complex semantics of Rust lifetimes make it hard to triage the reported warnings. Yuga provides a well-formatted HTML report with necessary information about the identified defect.

4.2.3 Effectiveness. There is no existing Rust embedded systems bug dataset. The situation is the same for C/C++ [129]. , which also contains references to the complete results We evaluated the effectiveness of SAST tools on our embedded crates dataset. The last column of Tbl. 5 has references to the complete results for each tool. *Robustness Issues:* SAST tools fail to handle the diverse build configurations, code structures, and semantics of embedded Rust crates. Consequently, these tools failed on several crates. The Tbl. 4 summarizes different classes of failures, affected tools, and crates. The majority of failures are because of *"Toolchain Incompatibilities", i.e.,* tools fail to identify the backend toolchain required by crates and consequently fail to analyze.

Precision: Given the large number of warnings, we used a random sampling method to analyze the precision of the tools. Specifically, we picked 30 crates with more downloads than the median across all the crates. This is to avoid selecting unimportant or rarely used crates.

We ignored rCanary and FFIChecker as their warnings did not contain enough information. Furthermore, even for other tools (*e.g.,* Lockbud), the information provided is not always sufficient to triage the corresponding warning. We categorized each warning into True Positive (TP), False Positive (FP) or Insufficient Information (IsIn). Tbl. 6 shows the results, the top two reasons for false positives, and the corresponding examples. First, tools were able to find real defects. Our Extended Report [126] shows a real deadlock found by Lockbud in the `tracing-log` crate. However, the true positive rate is very low. Contrary to tools' claim, all tools suffer from a very high false positive rate (40%-90%) on embedded crates. This is unsurprising as all these tools are evaluated (mostly) on non-embedded crates. This indicates that the design choices of the current tools fail to consider embedded crates.

Security Implications: Our results indicate that developers cannot solely rely on existing automated SAST tools to assess their crates and should also perform manual or semi-automated assessments.

4.3 C to Rust Conversion Tools

We selected C to Rust conversion tools by following the same approach as for SAST tools (§4.2.1). Although several tools satisfy

Table 6: Summary of manual analysis of results of various Rust SAST tools with True Positives (TP), False positives (FP), and Insufficient Information (IsIn). We list the top two reasons for FPs here (examples and complete results in our Extended Report [126]). Details in §4.2.3.

Tool Name	Analysis Results			Top 2 FP Reasons
	TP	FP	IsIn	
Lockbud	10 (33%)	14 (46%)	6 (20%)	Lock type ambiguity (42%) Complex Program Semantics (32%)
Rudra	1 (2.7%)	36 (97.2%)	N/A	Ignoring Explicit Guards (50%) Ignoring Atomic Types (30%)
Yuga	10 (33%)	13 (43%)	7 (23.3%)	Ignoring Caller Contexts (56%) Complex Program Semantics (30%)
SafeDrop	9 (30%)	17 (56.6%)	4 (13.3%)	Infeasible Paths (80%) Analysis Imprecision (10%)

Finding RQ1.4: Current SAST tools lack the necessary features required to effectively handle embedded crates.
Finding RQ1.5: Current SAST tools do not provide the necessary information to triage the reported defects, making it hard (rather impossible) to verify the reports.
Finding RQ1.6: Current SAST tools fail to effectively handle build idioms and configurations of embedded Rust crates, resulting in robustness issues.
Finding RQ1.7: The design choices of current SAST tools fail to effectively handle the common idioms in embedded crates resulting in a very high false positive rate (40%-90%).

Open Problem P1.5: There is no dataset of security bugs in Rust embedded crates. Recent systematic bug dataset creation works [59] provide possible approaches to tackle this.

our requirements, we present the results of only the c2rust tool. Other recent tools, such as laertes [50] and CRustS [78], do not work directly on C code but rather improve the Rust code produced by c2rust through novel post-processing techniques. As we will show in §4.3.3, c2rust either failed or produced uncompilable Rust code on (almost) the entire dataset. Consequently, recent tools that depend on c2rust also failed on the dataset.

4.3.1 Dataset. We collected popular C/C++ based RTOS from osrtos.com, which maintains the list of all popular RTOSes released to date. We selected well-maintained (*i.e.,* has build instructions) and compilable RTOSes. This resulted in a total of 16 C/C++ RTOSes (*CRT*). The compilation of RTOSes is specific to an MCU and includes HAL and other peripheral access libraries for the MCU. Thus, using RTOSes enables us to test the effectiveness of c2rust on codebases across different layers of embedded systems.

4.3.2 Running c2rust*.* To convert a project, we first need to capture compilation commands, *e.g.,* generating `compile_commands.json` using `scan-build` [81]. Next, we need to run c2rust on the captured `compile_commands.json`. c2rust uses clang to parse C files and uses pattern-based techniques on the resulting Abstract Syntax Tree (AST) to produce corresponding Rust code. Specifically, each compilation command (from `compile_commands.json`) will be executed by replacing the compiler with clang. However, just replacing the compiler will not work as embedded systems use non-standard and MCU specific toolchains, *e.g.,* `avr-gcc`, whose compiler flags/options may not be supported by clang. We followed an on-demand approach to convert into a clang compatible variant and run c2rust. Specifically, for each incompatible option leading to an error in conversion/compilation, we refer to clang's documentation to see if there is an alternative option (case-1), or if it is not supported by clang (case-2). For case-1, we use the corresponding alternative flags, *e.g.,* we replace `-march=nehalem` with `-march=armv8-a`. For case-2, we remove those flags/options (5 flags), *e.g.,* `-Wformat-overflow`. The removal of case-2 flags does not affect the conversion (a frontend task), as all of these flags are related to optimization (a middle/backend task).

4.3.3 Results. Our Extended Report [126] has a summary of the results. All RTOSes, except for two, required manually fixing `compile_commands.json` (discussed in §4.3.2). c2rust *failed on 6 (37.5%) RTOSes.* The two main reasons for this are: (i) Embedded system codebases often use (clang) unsupported C language features, and (ii) c2rust uses Rust `std` library to generate certain wrapper functions, but as mentioned in §2.2, `std` library should not be used in an embedded environment. For instance, `gnucc/oscore.c` file in `stateos/StateOS` uses parameter references in naked functions, which is not supported by clang [82] and consequently, c2rust fails. It executed successfully on 10 (62.5%) RTOSes. Out of which, *the generated* Rust *code was incorrect or syntactically invalid (e.g., missing semicolon) on 9 (90%) RTOSes.* The conversion was successful (*i.e.,* c2rust produced compilable Rust code) on only one RTOS, *i.e.,* `kmilo17pet/QuarkTS`.

Finally, c2rust uses a syntactic approach and consequently produces Rust code with mostly `unsafe` blocks. Although recent works [50] have tried to improve the situation, the progress is rather slow and requires more focused efforts.

Finding RQ1.8: C to Rust tools fail on most, *i.e.,* 93.8% (15/16), embedded codebases because of the prevalent use of special compiler flags and non-standard C language features.
Finding RQ1.9: C to Rust tools do not consider the `no_std` requirement and consequently will generate Rust code inapplicable for embedded systems.

5 RQ2: Interoperability of Rust

Most existing embedded system codebases are written in C [129]. Developers should be able to write Rust code that can interoperate with existing C code to avoid reengineering the entire embedded software stack in Rust. As mentioned in §2.2, Rust has Foreign Function Interface (FFI) support enabling interoperability with code written in other languages, especially C.

To answer this research question, we investigate the effort and challenges in developing Rust (or C) code that can interoperate with C (or Rust) code. We first provide a brief overview of recommended steps to develop interoperable code and quantify the effort and challenges specific to embedded systems. Second, we will present our experience and challenges in engineering interoperable code in various embedded system development scenarios.

5.1 Rust ⇔ C

The top part of Tbl. 7 summarizes our observations.

Table 7: Summary of Rust Interoperability Modes. We indicate whether each step is easy (🙂), (*e.g.,* running a tool on a C file), requires medium effort or *automation opportunities* (😐) (*e.g.,* configuring linker script), or requires significant effort or *open-problems* (☹) (*e.g.,* rewriting embedded C code in Rust). The challenges affecting embedded systems are highlighted.

Interoperability Modes			Method	Effort	Embedded System Specific Challenges
Mode	**Sub Abbr.**	**Desc.**			
R <-->C (§5.1)	R->C (§5.1.1)	Calling C function from Rust	1. Use bindgen to get declaration in Rust (🙂) 2. Link with target C object file (😐)	Easy (All C types are FFI Compatible)	N/A
	C->R (§5.1.2)	Calling Rust function from C	1. Use cbindgen to get declaration in C (☹). 2. Link with target Rust object file (😐).	Depends on the use of Rust types not compatible with C types. (i.e., FFI Incompatible types)	Most functions in Embedded crates use FFI incompatible types.
Interoperability in Embedded System Components (§5.2)	RoC (§5.2.2)	Developing Rust Application on top of C-RTOS	1. Use bindgen to get C-RTOS functions' declarations in Rust (🙂). 2. Modify the linker script (😐).	Easy (All C type are FFI compatible)	N/A
	CoR (§5.2.3)	Developing C Application on top of Rust RTOS	1. Use cbindgen to get Rust-RTOS functions' declarations in C (☹). 2. Modify the linker script (😐).	Depends on the use of FFI incompatible types in Rust RTOSes.	There is a prevalent use of FFI incompatible types in Rust RTOSes.
	RwC (§5.2.4)	Converting a component in C-RTOS to Rust	1. Use bindgen to convert all dependent component C headers to Rust (🙂). 2. Rewrite the target embedded component in Rust (☹). 3. Modify the Makefile (😐).	Depends on the effort to rewrite C code to Rust.	C to Rust conversion tools fail to handle embedded codebases, forcing manual rewriting.

5.1.1 Calling C function from Rust *(*Rust *→ C).* To invoke a C function from from Rust, first, we need to provide the Rust FFI signature of the function. This can be done using tools such as `bindgen` [24] to automatically generate FFI signatures from C header files. Then, they can link the library (*i.e.,* object file) containing the C function with the Rust object file to get the final executable. We illustrate these steps with an example in Our Extended Report [126]. One of the main tasks here is to generate FFI bindings for the C functions. It is relatively straightforward to create these bindings as the Rust's type system [84] is a superset of C's, *i.e.,* every builtin C type has a corresponding type in Rust. Finally, the target object file created from Rust code should be linked to the source C project. However, there are no automated tools to achieve this. In summary, it is relatively straightforward to write Rust code that can invoke C functions, but automation opportunities exist.

5.1.2 Calling Rust *function from C (C →* Rust*).* Similar to Rust → C (§5.1.1), here we need to generate C declaration for the target Rust function, which can be automated using `cbindgen` [87] tool (Our Extended Report [126] provides details of this process). The superior Rust type system has several types that are not supported in C. For instance, `Vec` [117], one of the most commonly used Rust types, is not supported in C. Consequently, `cbindgen` fails for such functions. Developers need to write type wrappers to handle this manually. But advanced features of Rust types, such as **`trait`** [137], makes engineering these wrapper functions challenging [61], more details in Our Extended Report [126]. We also performed a type compatibility analysis to assess the extent to which external functions in Rust crates use advanced Rust types, *i.e.,* library functions for which developers need to engineer corresponding type wrapper functions manually. Our Extended Report [126] provides details of the same. This is also the difficulty faced by developers (**RQ3.4**) as we discuss in §6.3.

Finding RQ2.1: Although it is relatively straightforward to invoke C functions from Rust code, automation opportunities exist to ease the process.
Finding RQ2.2: The use of FFI incompatible types makes it hard to invoke Rust functions from C code. The majority (~70%) of Rust embedded crates have functions with incompatible types.

Open Problem P2.1: Embedding rust function calls in C application is challenging due to the need for type conversion between C types and FFI-incompatible rust types. One possible approach is to manually create (once for all) type wrappers for basic complex types (*e.g.,* `Vec`) and use them to automatically create wrappers for composite types (*e.g.,* **`struct`**).

5.2 Rust Interoperable Challenges in Embedded Systems Development

We used Rust in various real-world scenarios to investigate this aspect. Specifically, we explore: Rust application on top of C RTOS (RoC), C application on top of Rust RTOS (CoR) and converting a component in C RTOS to Rust (RwC). The bottom part of Tbl. 7 summarizes our observations.

5.2.1 Setup. We chose the blinker application [95] for our application scenarios (RoC and CoR) as it encompasses all the necessary aspects of a typical embedded system, *i.e.,* interacts with RTOS, has event-driven custom interrupt handler, and uses call-backs. The application periodically (through an interrupt handler) blinks an LED by interacting through GPIO addresses. We used the `nrf52840-dk` MCU board [96] with `ARM Cortex-M4` for our target board, as it is a widely recognized and adopted development platform in the embedded systems community and is well-supported by Rust. We used FreeRTOS [52] as our C RTOSes, because of its widespread popularity in the embedded systems community [141] and extensive documentation [53]. As mentioned in §3.1, Rust RTOS can be either fully developed in Rust (*i.e.,* native) or wrappers around a C RTOS. We selected lilos [23] and FreeRTOS.rs [54] as our native and wrapper RTOSes, respectively. lilos is a stable and purely Rust based and completly asynchronous RTOS. This is a representative Rust based RTOS using the strongly suggested async design pattern [85].

5.2.2 Rust application on top of C RTOS (RoC). Our goal is to create a Rust blinky application on top of C FreeRTOS. We followed similar steps as described in §5.1.1. First, we generated embedded system compatible (*i.e.,* no_std) Rust FFI bindings from FreeRTOS header files using bindgen. Second, we developed blinky application using these FFI bindings. Our Extended Report [126] shows a snippet of creating a task using FreeRTOS through its FFI bindings. Specifically, we converted Rust types into appropriate FFI types and invoked the target function. We followed a similar procedure for all other steps, *i.e.,* registering interrupts, etc. Finally, we created a static library of C FreeRTOS and linked it with our Rust application to get the final executable. We tested the final executable and ensured that it worked as expected. *The entire process was straightforward.* The only issue was creating a linker script suitable for the target board. As mentioned before in §5.1.2, the availability of automated tools will make this process easier.

Listing 1: FFI incompatible function and FFI-friendly wrapper function to create tasks in Lilos scheduler

```
// FFI incompatible function
pub fn run_tasks(
futures: &mut [Pin<&mut dyn Future<Output = Infallible>>],
initial_mask: usize,
) -> !

#[no_mangle]
pub extern "C" fn lilos_run_two_tasks(fn1: *mut fn(), fn2: *mut
        fn(), initial_mask: usize) -> ! {
  unsafe {
     let fut1 = *fn1;
     let future1 = pin!(async move {
         loop { fut1() } });
     let fut2 = *fn2;
     let future2 = pin!(async move {
         loop { fut2() } });
     run_tasks(&mut [future1, future2], initial_mask);
  }
}
```

5.2.3 C application on top of Rust RTOS (CoR). This interoperable modality is crucial for developers who seek to build secure systems by leveraging existing components. Furthermore, as shown in Fig. 4, 36% of developers claim to have developed C code calling Rust functions. Here, our goal is to create a C blinky application on top of Rust RTOSes, specifically on FreeRTOS.rs (Rust wrapper of C FreeRTOS) and lilos (a pure Rust RTOS). We followed similar steps as described in §5.1.2.

- *On* FreeRTOS.rs*:* Being a wrapper, all external functions used C compatible types, and cbindgen was able to create C declarations for all the required functions. This made it easy to create the main task of the C blinky application. However, accessing GPIO pins required us to use nrf52840_pac [116] Rust create, which uses a C incompatible type, *i.e.,* RegisterBlock. Consequently, cbingen failed to create corresponding C declarations. We manually created an FFI compatible Rust function (togglePin) to access GPIO pins and used it in our application. Refer our Extended Report [126] for details.
- *On* lilos*:* This presented an extreme case wherein none of the external functions are FFI compatible, and consequently, cbindgen failed to create C declarations. We had to manually create FFI compatible wrapper functions (*e.g.,* lilos_run_two_tasks for run_tasks in Lis. 1). For accessing GPIO pins, we followed the same approach as described before in *On* FreeRTOS.rs.

The main challenge in both cases was dealing with incompatible Rust types. We found (from our analysis in §5.1.2) that on-average of 26 interface functions in Rust RTOSes use incompatible Rust types.

Finding RQ2.3: Significant development effort is required to engineer a C-embedded application on top of Rust RTOSes because of the prevalent use of incompatible Rust types.

5.2.4 Rust component in C RTOS (RwC). Here, we aim to convert a component in C RTOS into Rust to mimic an incremental porting scenario. We selected list component in C FreeRTOS, as it is self-contained (*i.e.,* no calls to other components). We followed a similar procedure as described in §5.1.1. First, we used bindgen on list.h to create the required Rust types. The xLIST (in our Extended Report [126]) shows the type generated by bindgen. Second, we reimplemented the list functions (in list.rs) using the types generated by bindgen. Unfortunately, as mentioned in §4.3, the recommended way to convert C to Rust code does not work on embedded codebases. We manually translated the corresponding C implementation line-by-line into Rust, which required considerable effort. Our Extended Report shows a snippet of vListInitialise function in Rust. Finally, we modified the Makefile to build list.rs into a static library and linked it with the final FreeRTOS object file.

Finding RQ2.4: The lack of embedded codebase support in C-to-Rust conversion tools (described in §4.3) poses a considerable challenge in adopting the (recommended) incremental porting approach [50, 134] to convert embedded codebases to Rust.

6 RQ3: Developers Perspective

We aim to shed light on developers' perspectives on using Rust for embedded systems development. Specifically, (i) Reasons for not using Rust.; (ii) Challenges faced by developers in using Rust.; and (iii) Developer's perspective on Rust's performance, safety and interoperability.

6.1 Study Methodology

We used an anonymous online survey with questions spanning various categories as shown in Tbl. 8. We recruited participants by sending the link to our survey to various embedded systems communities and Rust embedded developers' mailing lists (Details in Our Extended Report [126]). Also, we used our industry collaborations to circulate our survey to multiple organizations. Our Institutional Review Board (IRB) reviewed and approved our study protocol.

Survey Respondents. We got 268 responses, out of which we filtered out 43 responses from inattentive participants (through

Table 8: Summary of Survey Questions. Exact questions in Our Extended Report [126]

Category	Description	No. of questions
Familiarity and Experience	Examines participants' familiarity, experience, and preferred languages for embedded systems development.	9
Acquaintance with Rust	Explores familiarity with Rust and its specific features of participants.	15
Reasons to Use Rust	Gathers opinions of participants on reasons to use Rust, its advantages, and perceived challenges.	13
Hardware Support, Integration, and Performance	Enquires about issues related to hardware support, integration, and performance when using Rust for embedded development.	11
Memory Safety and Debugging	Focuses on the importance of memory safety, ease of debugging with Rust, and related practices in embedded systems development.	9
Documentation and Community Support	Evaluates the quality of Rust documentation and the level of community support available for embedded systems.	6
Development Time and Code Quality	Investigates views on potential gains in development time and code quality when using Rust.	3

attention-checking questions), resulting in 225 valid responses. There is considerable diversity in the embedded systems experience of participants, indicating a representative developers group. Our Extended Report shows the distribution of embedded systems experience of participants.

6.2 Not Using Rust for Embedded Systems: Expectations v/s Reality

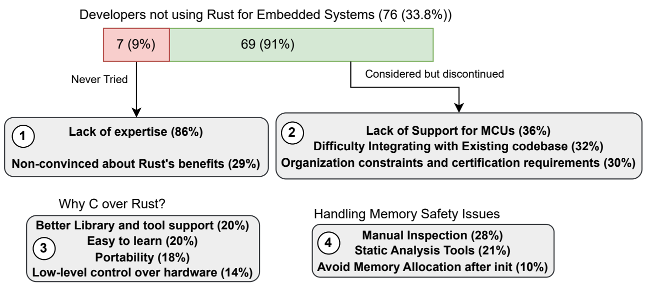
Developers not using Rust for Embedded Systems (76 (33.8%))
7 (9%)
69 (91%)
Never Tried
Considered but discontinued
1 Lack of expertise (86%)
Non-convinced about Rust's benefits (29%)
2 Lack of Support for MCUs (36%)
Difficulty Integrating with Existing codebase (32%)
Organization constraints and certification requirements (30%)
Why C over Rust?
Better Library and tool support (20%)
3 Easy to learn (20%)
Portability (18%)
Low-level control over hardware (14%)
Handling Memory Safety Issues
4 Manual Inspection (28%)
Static Analysis Tools (21%)
Avoid Memory Allocation after init (10%)

Figure 1: Response summary of Developers not using Rust.

The Fig. 1 shows the summary of 76 (33.8%) participants who currently do not use Rust. Only, 7 (9%) participants never tried to use Rust, mainly because of the lack expertise (①). Furthermore, 29% of developers are not convinced about Rust security benefits as *embedded systems rarely use dynamic memory allocation and do not need Rust's ownership features — an important safety feature of* Rust.

However, the other 69 (91%) participants considered Rust, but discontinued because of three main reasons (②): (i) Lack of support for MCUs, this is inline with our analysis in §4. (ii) Integrating with existing codebases. (iii) Organizational and certification constraints. Source code used as part of critical infrastructure, such as airplanes, undergo rigorous certification [47, 69, 70]. This is expensive and time-consuming. Switching to Rust requires re-certification, which may not be desirable for organizations.

All developers in Fig. 1 use C, and the ③ box shows the reasons for choosing C. The first two reasons are expected, as C is an old language with many libraries and toolchain support. The third reason, *i.e.,* Portability, is interesting. In C, there are no language-specific considerations for embedded systems. Consequently, it is relatively easy to port (or repurpose) an existing library for the embedded use case by linking it with embedded versions of standard libraries. However, in Rust, *embedded libraries (i.e., crates) should be developed with* `no_std` *environment — which restricts the uses of certain language-level features. Consequently, porting existing libraries to be* `no_std` *compatible and to use in embedded systems is challenging [40, 46].*

Interestingly, as shown in ④ of Fig. 1, many (28%) embedded systems developers (using C) do not use any automated security tools and rely on manual inspection. Only 21% of the developers use static analysis tools. This confirms observations made by a recent study [129]. Finally, none of the developers use any dynamic analysis tools.

Finding RQ3.1: To improve adoption of Rust for embedded systems:
- Support needs to be added for more MCUs.
- Techniques and methods should be developed to ease the certification of Rust code ported from already certified C code.
- Automated techniques should be developed to convert Rust crates to `no_std` compatible.

6.3 Experiences in Using Rust for Embedded Systems

There were 149 (66.2%) participants who currently use Rust for embedded systems development. These participants have varied development experience with Rust, specifically, 19% with < 6 months, 28% with 6 months - 1 year and 18% with 1-2 years, and 35% with more than 2 years.

Adopting Rust and Motivation: The two main motivations to learn Rust for embedded systems are safety and reliability (94%) and familiarity with the language (57%). Although there exists good support for Rust in the embedded systems community (Fig. 7), the majority percentage (85%) of developers claim that it still requires considerable effort (*i.e.,* Moderate (40%) + Hard (45%)) to adopt Rust.

Rust Documentation and Community Support (Fig. 6 and 2): The majority, *i.e.,* 81% (51 + 30) of developers, agree that the available documentation and community are helpful. However, 49% (30 + 19) of developers mention that documentation should be improved. The Fig. 2 shows specific suggestions to improve the documentation. Specifically, Rust *documentation should contain more examples and be organized better.*

Developer Tools and Crates: 68% of the developers think the currently available crates provide sufficient support (*i.e.,* very satisfied – somewhat satisfied), whereas the rest, *32%, observe that it is not*

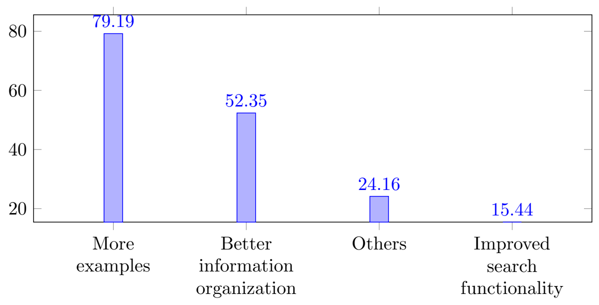

Figure 2: Required Improvements to Rust Documentation.

adequate (i.e., dissatisfied – very dissatisfied). This is also in line with our analysis (§4), where we noticed that necessary support (*i.e.,* HAL and other necessary crates) is unavailable for certain MCUs.

Fig. 3 shows the opinion of users w.r.t the Rust toolchain support. Most developers (across all experience levels) mention that it is easy to adopt Rust toolchain for embedded systems development — reasons are intuitive tools and their documentation (⑤). This is in line with our analysis in §5. Nonetheless, 12% (18) developers expressed concerns, *i.e.,* poor documentation (missing examples) and buggy tools (⑥). These could be because of using tools from non-stable branches. This also further confirms observations in Fig. 2, where developers require more examples to be included in the documentation.

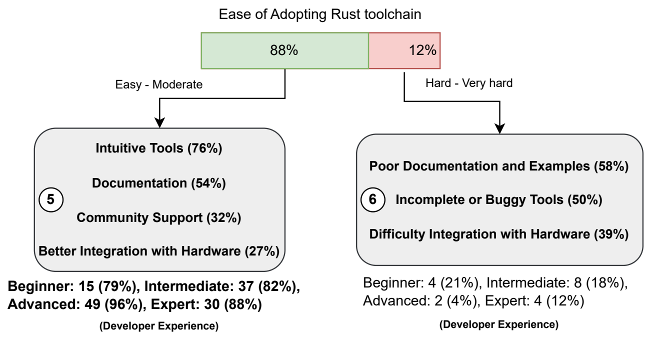
Ease of Adopting Rust toolchain
88%
12%
Easy - Moderate
Hard - Very hard
5
Intuitive Tools (76%)
Documentation (54%)
Community Support (32%)
Better Integration with Hardware (27%)
6
Poor Documentation and Examples (58%)
Incomplete or Buggy Tools (50%)
Difficulty Integration with Hardware (39%)
Beginner: 15 (79%), Intermediate: 37 (82%), Advanced: 49 (96%), Expert: 30 (88%)
(Developer Experience)
Beginner: 4 (21%), Intermediate: 8 (18%), Advanced: 2 (4%), Expert: 4 (12%)
(Developer Experience)

Figure 3: Response summary of Developers on ease of adopting Rust toolchain.

Performance of Rust: It is interesting to see that only 54% of developers mentioned that they performed a systematic comparative evaluation of their Rust implementation with existing C implementation. Wherein 28.5% noticed similar performance, 22% noticed that Rust was faster, and the remaining 3.5% noticed that Rust implementation was slower.

The slowdown observations contradict the common belief that given the asynchronous nature of embedded systems, the performance of Rust's implementation can be significantly improved by carefully using its built-in features, such as closures [60]. These observations also highlight *the need for a systematic performance evaluation of using* Rust *for embedded systems.*

Finding RQ3.2: Rust documentation should be improved with more embedded system-specific examples.

Open Problem P3.1: Developers have contradictory views on Rust's performance on embedded systems. Existing performance studies [21, 71, 144] could be extended to include Rust embedded systems.

Interoperability with Existing Codebase: All developers agree that interoperability is needed, and most developers (98%) were aware of Rust's interoperability support. However, 56% of developers mentioned that they face challenges in using interoperability support of Rust. The ⑦ box in Fig. 4 shows developers' common challenges in using interoperability support.

Finding RQ3.3: The majority (*i.e.,* 34%) of developers face issues handling type incompatibilities between Rust and C code. This is in line with our analysis (§5.1.2), where we show that handling data types is one of the challenges in using C on top of Rust (CoR).

The second major (26%) issue is debugging, which is expected because, as explained in §2, embedded systems follow an asynchronous and event-driven design. This results in frequent cross-language domain interactions and makes debugging hard.

As shown by ⑧ in Fig. 4, 60% (32 + 28) of developers agree that using interoperable Rust improves security. However, 32% mention that secure usage (*i.e.,* through `unsafe` blocks) requires significant effort — which is in line with existing works [17] that show that engineering interoperable code in `unsafe` blocks is challenging and prone to security issues.

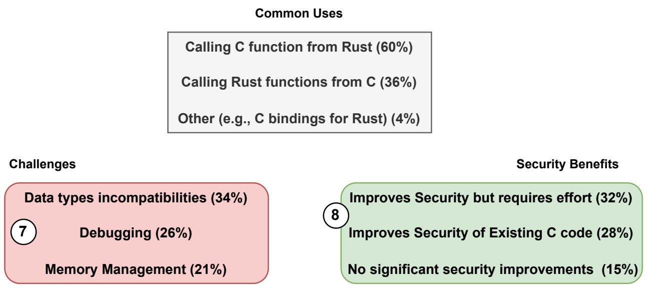
Common Uses
Calling C function from Rust (60%)
Calling Rust functions from C (36%)
Other (e.g., C bindings for Rust) (4%)
Challenges
7
Data types incompatibilities (34%)
Debugging (26%)
Memory Management (21%)
Security Benefits
8
Improves Security but requires effort (32%)
Improves Security of Existing C code (28%)
No significant security improvements (15%)

Figure 4: Response summary of Developers perspectives on Rust's Interoperability.

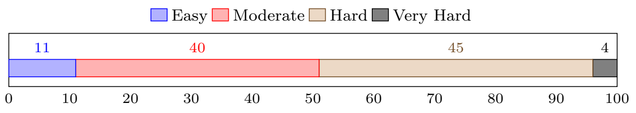

Figure 5: Ease of Adopting Rust for Embedded Systems Development

Rust **v/c C:** 92% of developers mentioned that they also used C for embedded systems development. Out of which, *64% of developers claim that development time significantly decreased and also the code quality improved after switching to* Rust. This is in line with recent findings at Google [31]. For embedded systems development, 30% of developers recommend Rust unconditionally, whereas *61% recommend* Rust *only if the developer is well-versed in it*, and the 9% recommend Rust only if safety is of high importance.

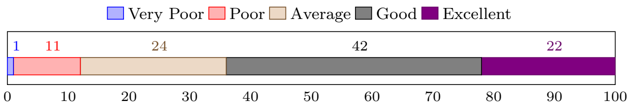

Figure 6: Support for Rust in Embedded Systems Community

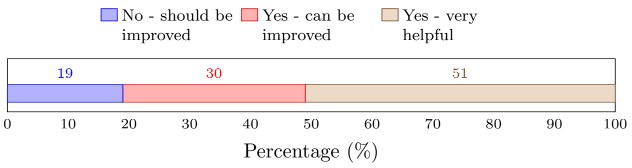

Figure 7: Is Rust Documentation and Community Helpful?

7 Limitations and Threats to Validity

We acknowledge the following limitations and threats to validity:

- Our findings are based on the analysis of the collected dataset and SAST tools. The dataset and the tools may not be representative enough. We tried to avoid this by collecting crates and tools from diverse sources.
- We did not analyze all the alerts raised by various SAST tools (§4.2). Consequently, using these alerts to assess the quality of crates could be exaggerated because of potential false positives.
- Our RQ2 (§5.2) observations are based on limited scenarios and may not be generalizable. However, the developer survey (in §6.3) confirmed our findings, reducing the risk.

8 Related Work

Rust Studies: Several works [17, 55, 88, 103, 145] evaluate various aspects of Rust from the usability perspective. Fulton *et al.,* [55] surveyed and interviewed Rust developers to understand challenges and barriers to adoption. Similarly, Zeng *et al.,* [145] performed a mixed-methods study of Rust related forums to identify common challenges and corresponding solutions. However, other works assess specific aspects of Rust. Astrauskas *et al.,* [17] focused on identifying common uses of `unsafe` blocks. Whereas, Qin *et al.,* [103] focused on identifying challenges in using concurrency constructs and identified common causes of concurrency issues in Rust code. Similarly, Mindermann *et al.,* [88] exclusively studied the usability of Rust's cryptography APIs, providing crucial recommendations for developing these APIs to enhance usability and reduce misuse.

Pinho *et al.,* [100] and Ashmore *et al.,* [16] evaluated the feasibility of using Rust for safety critical systems (a class of embedded systems). Specifically, using evaluation criteria for programming languages, aligning with the standards set by RTCA DO-178C, they demonstrated that Rust meets all the criteria. Levy *et al.,* [74] focused on using Rust for kernel development and shared their first-hand experience in creating a kernel for low-power MCUs. They also demonstrated [73, 75] the feasibility of using Rust to engineer common kernel building blocks with only a few `unsafe` blocks. This paper assesses the applicability and challenges of using Rust for embedded system software, such as RTOSes, by performing a systematic analysis and developer study.

Embedded Systems Vulnerabilities: Several works [49, 51, 57] try to understand vulnerabilities in embedded systems and analyze challenges and possible solutions for effective vulnerability detection. Several embedded systems vulnerability detection techniques use various approaches ranging from static analysis [105], symbolic execution [33], and rehosting-based dynamic analysis [125] or fuzzing [29]. In this work, we do not propose any new techniques but rather use state-of-the-art tools (§4.2) to assess various aspects of Rust embedded software.

9 Conclusion

We performed a systematic analysis and a comprehensive (with 225 developers) survey to understand the current state and challenges in using Rust for embedded systems development. Our findings provide insights into the current state and expose open problems and potential improvements that can facilitate easy adoption of Rust for embedded system development.

10 Acknowledgements

This research was supported in part by the National Science Foundation (NSF) under Grant CNS-2340548, Rolls-Royce Grant on "Dynamic Analysis of Embedded Systems", and Defense Advanced Research Projects Agency (DARPA) under contract numbers N660012 0C4031 and N660012224037. The U.S.Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF, Rolls-Royce, or the United States Government.

References

[1] Conditional compilation - The Rust Reference. https://doc.rust-lang.org/reference/conditional-compilation.html.
[2] How Rust is Made and "Nightly Rust" - The Rust Programming Language. https://doc.rust-lang.org/book/appendix-07-nightly-rust.html.
[3] SARIF Home. https://sarifweb.azurewebsites.net/.
[4] The 5 Worst Examples of IoT Hacking and Vulnerabilities in Recorded History. https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities, June 2020.
[5] Vali Kh Abdrakhmanov, Niaz N Bikbaev, and Renat B Salikhov. Development of low-cost electronic training boards based on universal microcontroller. In *2016 13th International Scientific-Technical Conference on Actual Problems of Electronics Instrument Engineering (APEIE)*, volume 1, pages 319–325. IEEE, 2016.
[6] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Djamel Eddine Khelladi, and Jean-Marc Jézéquel. *Learning from thousands of build failures of Linux kernel configurations.* PhD thesis, Inria; IRISA, 2019.
[7] Abdullah Al-Boghdady, Khaled Wassif, and Mohammad El-Ramly. The presence, trends, and causes of security vulnerabilities in operating systems of iot's low-end devices. *Sensors*, 21(7), 2021.
[8] Mohammed Ali Al-Garadi, Amr Mohamed, Abdulla Khalid Al-Ali, Xiaojiang Du, Ihsan Ali, and Mohsen Guizani. A Survey of Machine and Deep Learning Methods for Internet of Things (IoT) Security. *IEEE Communications Surveys & Tutorials*, 22(3):1646–1685, 2020.
[9] Fadi Al-Turjman and Joel Poncha Lemayian. Intelligence, security, and vehicular sensor networks in internet of things (iot)-enabled smart-cities: An overview. *Computers & Electrical Engineering*, 87:106776, 2020.
[10] Hussain M. J. Almohri and David Evans. Fidelius charm: Isolating unsafe rust code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 248–255, New York, NY, USA, 2018.
[11] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. SoK: Security Evaluation of Home-Based IoT Deployments. *IEEE Symposium on Security and Privacy (SP)*, 2019-May:1362–1380, 2019.
[12] Amit Seal Ami, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. " false negative–that one is going to kill you": Understanding industry perspectives of static analysis based security testing. *arXiv preprint arXiv:2307.16325*, 2023.
[13] AMNESIA:33 – Foresout Research Labs Finds 33 New Vulnerabilities in Open Source TCP/IP Stacks. https://www.forescout.com/blog/amnesia33-forescout-research-labs-finds-33-new-vulnerabilities-in-open-source-tcp-ip-stacks/, December 2020.
[14] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th USENIX Security Symposium*, pages 1093–1110, 2017.
[15] URGENT/11. https://www.armis.com/research/urgent11/.
[16] Rob Ashmore, Andrew Howe, Rhiannon Chilton, and Shamal Faily. Programming language evaluation criteria for safety-critical software in the air domain.

In *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 230–237, 2022.
[17] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.
[18] Azure RTOS. https://github.com/azure-rtos.
[19] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 84–99, 2021.
[20] Inyoung Bang, Martin Kayondo, Hyungon Moon, and Yunheung Paek. Trust: A compilation framework for in-process isolation to protect safe rust against untrusted code. In *32nd USENIX Security Symposium*, 2023.
[21] Rust vs C++ g++ - Which programs are fastest? https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html.
[22] Deval Bhamare, Maede Zolanvari, Aiman Erbad, Raj Jain, Khaled Khan, and Nader Meskin. Cybersecurity for industrial control systems: A survey. *Computers & Security*, 89:101677, 2020.
[23] Cliff L. Biffle. lilos: A minimal async rtos. https://github.com/cbiffle/lilos, 2023.
[24] bindgen - Rust. https://docs.rs/bindgen/latest/bindgen/.
[25] BlueHat. Memory corruption is still the most prevalent security vulnerability. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues, 2019.
[26] Carsten Bormann, Mehmet Ersue, and Ari Keränen. Terminology for Constrained-Node Networks. Request for Comments RFC 7228, Internet Engineering Task Force, May 2014.
[27] William Bugden and Ayman Alahmar. Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503*, 2022.
[28] Archana Chaudhary, Savita Kolhe, and Raj Kamal. An improved random forest classifier for multi-class classification. *Information Processing in Agriculture*, 3(4):215–222, 2016.
[29] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, Xiaofeng Wang, W. Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Network and Distributed System Security Symposium*, 2018.
[30] LLVM CFI and Cross-Language LLVM CFI Support for Rust. https://bughunters.google.com/blog/4805571163848704/llvm-cfi-and-cross-language-llvm-cfi-support-for-rust.
[31] Thomas Claburn. Rust developers at Google twice as productive as C++ teams. https://www.theregister.com/2024/03/31/rust_google_c/.
[32] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C Necula. Dependent types for low-level programming. In *Proceedings of the 2007 European Symposium on Programming (ESOP)*, pages 520–535. Springer, 2007.
[33] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In *27th USENIX Security Symposium*, page 309–326, 2018.
[34] arduino_nano_connect - crates.io. https://crates.io/crates/arduino_nano_connect.
[35] atdf2svd crates.io. https://crates.io/crates/atdf2svd.
[36] crates.io: Rust Package Registry. https://crates.io/.
[37] eeprom24x - crates.io. https://crates.io/crates/eeprom24x.
[38] embedded-hal - crates.io. https://crates.io/crates/embedded-hal.
[39] futures-executor - crates.io. https://crates.io/crates/futures-executor.
[40] no-std-compat - crates.io. https://crates.io/crates/no-std-compat.
[41] oc-wasm-futures - crates.io. https://crates.io/crates/oc-wasm-futures.
[42] svd2rust - crates.io. https://crates.io/crates/svd2rust.
[43] tinybmp - crates.io. https://crates.io/crates/tinybmp.
[44] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Transactions on Software Engineering and Methodology*, 2021.
[45] Mohan Cui, Suran Sun, Hui Xu, and Yangfan Zhou. rcanary: Detecting memory leaks across semi-automated memory management boundary in rust, 08 2023.
[46] Item 33: Consider making library code no_std compatible - Effective Rust. https://www.lurklurk.org/effective-rust/no-std.html.
[47] Ian Dodd and Ibrahim Habli. Safety certification of airborne software: An empirical study. *Reliability Engineering & System Safety*, 98(1):7–23, 2012.
[48] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 2016 International Conference on Compiler Construction (CC)*, pages 132–142. ACM, 2016.
[49] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecurity*, 5, 09 2022.
[50] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. Translating c to safer rust. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.
[51] Xiaotao Feng, Xiaogang Zhu, Qing-Long Han, Wei Zhou, Sheng Wen, and Yang Xiang. Detecting vulnerability on iot device firmware: A survey. *IEEE/CAA Journal of Automatica Sinica*, 10(1):25–41, 2023.
[52] FreeRTOS. https://www.freertos.org/index.html.
[53] Freertos book. https://www.freertos.org/Documentation/RTOS_book.html.
[54] Freertos-rust. https://github.com/lobaro/FreeRTOS-rust, 2023.
[55] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael W. Hicks, and Michelle L. Mazurek. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Symposium on Usable Privacy and Security*, 2021.
[56] Dharmalingam Ganesan, Mikael Lindvall, Rance Cleaveland, Raoul Jetley, Paul Jones, and Yi Zhang. Architecture reconstruction and analysis of medical device software. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 194–203. IEEE, 2011.
[57] Imran Ghafoor, Imran Jattala, Shakeel Durrani, and Ch Muhammad Tahir. Analysis of openssl heartbleed vulnerability for embedded systems. In *17th IEEE International Multi Topic Conference 2014*, pages 314–319, 2014.
[58] Merve Gülmez, Thomas Nyman, Christoph Baumann, and Jan Tobias Mühlberg. Friend or foe inside? exploring in-process isolation to maintain memory safety for unsafe rust. In *2023 IEEE Secure Development Conference (SecDev)*, pages 54–66, 2023.
[59] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), nov 2020.
[60] Hugo Heyman and Love Brandefelt. A comparison of performance & implementation complexity of multithreaded applications in rust, java and c++, 2020.
[61] The Rust FFI Omnibus. https://jakegoulding.com/rust-ffi-omnibus/.
[62] Ripple20. https://www.jsof-tech.com/disclosures/ripple20/.
[63] System View Description. https://www.keil.com/pack/doc/CMSIS/SVD/html/index.html.
[64] Samuel C Kendall. Bcc: Runtime checking for c programs. In *USENIX Summer Conference, 1983*, pages 5–16, 1983.
[65] Arslan Khan, Dongyan Xu, and Dave Jing Tian. Ec: Embedded systems compartmentalization via intra-kernel isolation. In *IEEE Symposium on Security and Privacy (SP)*, pages 2990–3007, 2023.
[66] Arslan Khan, Dongyan Xu, and Dave Jing Tian. Low-cost privilege separation with compile time compartmentalization for embedded systems. In *IEEE Symposium on Security and Privacy (SP)*, pages 3008–3025, 2023.
[67] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, page 132–148, 2022.
[68] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.
[69] Andrew Kornecki and Janusz Zalewski. Certification of software for real-time safety-critical systems: state of the art. *Innovations in Systems and Software Engineering*, 5:149–161, 2009.
[70] Andrew J Kornecki. Airborne software: communication and certification. *Scalable Computing: Practice and Experience*, 9(1), 2008.
[71] Speed of Rust vs C. https://kornel.ski/rust-c-speed.
[72] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS)*, page 51–57, 2017.
[73] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: Experiences building an embedded os in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS)*, page 21–26, 2015.
[74] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys)*, 2017.
[75] Amit A. Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Alexander Levis. Multiprogramming a 64kb computer safely and efficiently. *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.
[76] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. Mirchecker: Detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 2183–2196, 2021.
[77] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Detecting cross-language memory management issues in rust. In *European Symposium on Research in Computer Security*, pages 680–700. Springer, 2022.
[78] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. In rust we trust – a transpiler from unsafe c to safer rust. In *IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 354–355, 2022.
[79] Rust — The Linux Kernel documentation. https://www.kernel.org/doc/html/next/rust/index.html.
[80] Peiming Liu, Gang Zhao, and Jeff Huang. Securing unsafe rust programs with xrust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, pages 234–245, 2020.
[81] intercept-build · llvm-mirror/clang. https://github.com/llvm-mirror/clang/blob/master/tools/scan-build-py/bin/intercept-build.
[82] r217200 - Don't allow inline asm statements to reference parameters in naked functions. https://lists.llvm.org/pipermail/cfe-commits/Week-of-Mon-20140901/114154.html.
[83] Joel Margolis, Tae Tom Oh, Suyash Jadhav, Young Ho Kim, and Jeong Neyo Kim. An in-depth analysis of the mirai botnet. In *2017 International Conference on Software Security and Assurance (ICSSA)*, pages 6–12. IEEE, 2017.
[84] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.
[85] OpenSystems Media. Asynchronous event-driven architecture for high-reliability systems - military embedded systems. https:

//militaryembedded.com/radar-ew/rugged-computing/asynchronous-event-driven-architecture-for-high-reliability-systems.
[86] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *Proceedings of the 2022 Network and Distributed System Security Symposium (NDSS)*, volume 22, pages 1–17, 2022.
[87] Generating a Header File - The (unofficial) Rust FFI Guide. https://michael-f-bryan.github.io/rust-ffi-guide/cbindgen.html.
[88] Kai Mindermann, Philipp Keck, and Stefan Wagner. How usable are rust cryptography apis? *CoRR*, abs/1806.04929, 2018.
[89] MITRE. 2021 CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html, 2021.
[90] Krunal A Moharkar, Ankita A Tiwari, Pratik N Bhuyar, Pradip K Bedre, and FSA Bachwani. Review on different microcontroller boards used in iot. *Journal For Research in Applied Science and Engineering Technology*, 10:2321–9653, 2022.
[91] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*, 2018.
[92] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 245–258, 2009.
[93] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.
[94] Vikram Nitin, Anne Mulhern, Sanjay Arora, and Baishakhi Ray. Yuga: Automatically detecting lifetime annotation bugs in the rust language. *ArXiv*, abs/2310.08507, 2023.
[95] nrf interrupt application. https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v15.0.0%2Fpin_change_int_example.html.
[96] nrf52840-dk mcu board. https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK.
[97] Eoin O'driscoll and Garret E O'donnell. Industrial power and energy metering–a state-of-the-art review. *Journal of Cleaner Production*, 41:53–64, 2013.
[98] OSRTOS. https://www.osrtos.com/.
[99] Wanrong Ouyang and Baojian Hua. Rusbox: Towards efficient and adaptive sandboxing for rust. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 1–2, 2021.
[100] André Pinho, Luis Couto, and José Oliveira. Towards rust for critical systems. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 19–24, 2019.
[101] PYPL PopularitY of Programming Language index. https://pypl.github.io/PYPL.html.
[102] Dipika Roy Prapti, Abdul Rashid Mohamed Shariff, Hasfalina Che Man, Norulhuda Mohamed Ramli, Thinagaran Perumal, and Mohamed Shariff. Internet of things (iot)-based aquaculture: An overview of iot application on water quality monitoring. *Reviews in Aquaculture*, 14(2):979–992, 2022.
[103] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 763–779, 2020.
[104] Idiomatically convert to no-std. https://www.reddit.com/r/rust/comments/10f3nvn/how_do_you_idiomatically_convert_libs_to_no_std/.
[105] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. *IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561, 2020.
[106] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping safe rust safe with galeed. In *Annual Computer Security Applications Conference (ACSAC)*, page 824–836, 2021.
[107] Muhammad Danish Roshaidie, William Pang Han Liang, Calvin Goh Kai Jun, Kok Hong Yew, et al. Importance of secure software development processes and tools for developers. *arXiv preprint arXiv:2012.15153*, 2020.
[108] Android Rust Introduction. https://source.android.com/docs/setup/build/rust/building-rust-modules/overview.
[109] The AVR-Rust Guidebook. https://book.avr-rust.com/.
[110] Introduction - The Cargo Book. https://doc.rust-lang.org/cargo/.
[111] aarch64-cpu. https://github.com/rust-embedded/aarch64-cpu.
[112] Rust Embedded. https://github.com/rust-embedded.
[113] Awesome embedded Rust. https://github.com/rust-embedded/awesome-embedded-rust, June 2023. original-date: 2018-04-01T21:17:15Z.
[114] esp-hal. https://github.com/esp-rs/esp-hal.
[115] Lifetimes - Rust By Example. https://doc.rust-lang.org/rust-by-example/scope/lifetime.html.
[116] nrf52840pac - rust. https://docs.rs/nrf52840-pac/latest/nrf52840_pac/index.html.
[117] std::vec - Rust. https://doc.rust-lang.org/std/vec/index.html.
[118] Understanding Ownership - The Rust Programming Language. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.
[119] Unsafe Rust. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.
[120] References and Borrowing. https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html.
[121] The Rust Programming Language - The Rust Programming Language. https://doc.rust-lang.org/book/.
[122] e310x-hal. https://github.com/riscv-rust/e310x-hal.
[123] riscv-rust/hifive1: Board support crate for HiFive1 and LoFive boards. https://github.com/riscv-rust/hifive1.
[124] Aditya Saligrama, Andrew Shen, and Jon Gjengset. A Practical Analysis of Rust's Concurrency Story. *arXiv preprint arXiv:1904.12210*, 2019.
[125] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Reza Abbasi. Fuzzware: Using precise mmio modeling for effective firmware fuzzing. In *31st USENIX Security Symposium*, 2022.
[126] Ayushi Sharma, Shashank Sharma, Santiago Torres-Arias, and Aravind Machiry. Rust for embedded systems: Current state, challenges and open problems (extended report). *arXiv.org*, (arXiv:2311.05063v2), 2023.
[127] Shashank Sharma, Ayushi Sharma, and Aravind Machiry. Aunor: Converting rust crates to [no_std] at scale. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy (CODASPY)*, page 163–165, 2024.
[128] Mingjie Shen, James C Davis, and Aravind Machiry. Towards automated identification of layering violations in embedded applications (wip). In *2023 ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2023.
[129] Mingjie Shen, Akul Pillai, Brian A Yuan, James C Davis, and Aravind Machiry. An empirical study on the use of static analysis tools in open source embedded software. *arXiv preprint arXiv:2310.00205*, 2023.
[130] NB Soni and Jaideep Saraswat. A review of iot devices for traffic management system. In *2017 international conference on intelligent sustainable systems (ICISS)*, pages 1052–1055. IEEE, 2017.
[131] Joseph L Steffen. Adding run-time checking to the portable c compiler. *Software: Practice and Experience*, 22(4):305–316, 1992.
[132] K Swathi, T Uday Sandeep, and A Roja Ramani. Performance analysis of microcontrollers used in iot technology. *International journal of scientific research in science, engineering and technology*, 4(4):1268–1273, 2018.
[133] TOML: Tom's Obvious Minimal Language. https://toml.io/en/.
[134] An Empirical Study of C to Rust Transpilers.
[135] CVE Trends. Cve trends. https://www.cvedetails.com/vulnerabilities-by-types.php, 2021. Accessed: 2020-10-11.
[136] Margus Välja, Matus Korman, and Robert Lagerström. A study on software vulnerabilities and weaknesses of embedded systems in power networks. In *Proceedings of the 2nd Workshop on Cyber-Physical Security and Resilience in Smart Grids*, pages 47–52, 2017.
[137] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 321–330, 2022.
[138] Wemo WiFi Light Switch Smart Dimmer | Belkin: US. https://www.belkin.com/wifi-smart-dimmer/WDS060.html.
[139] Elecia White. *Making Embedded Systems: Design Patterns for Great Software*. "O'Reilly Media, Inc.", October 2011.
[140] Press Release: Future Software Should Be Memory Safe | ONCD. https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/, February 2024.
[141] The Express Wire. Real-time operating systems (rtos) market 2023: Research, growth and trends. https://www.benzinga.com/pressreleases/23/09/34197565/real-time-operating-systems-rtos-market-2023-research-growth-and-trends-industry-forecast-2030.
[142] Geeta Yadav and Kolin Paul. Architecture and security of scada systems: A review. *International Journal of Critical Infrastructure Protection*, 34:100433, 2021.
[143] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering (FSE)*, 1:1585–1608, 2024.
[144] Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.
[145] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. Learning and programming challenges of rust: A mixed-methods study. In *Proceedings of ACM/IEEE 44th International Conference on Software Engineering (ICSE)*, pages 1269–1281, 2022.