Static and Dynamic Memory Protection using Type-Based Memory Layout in Rust

Per Lindgren[*†], Pawel Dzialo[*], Antti Nurmi[†] and Henri Lunnikivi[†]
[*]Luleå University of Technology
Email:per.lindgren@ltu.se, pawdzi-7@student.ltu.se
[†]Tampere University
Email: henri.lunnikivi@tuni.fi, antti.nurmi@tuni.fi

Abstract—**Hardware supported memory protection is a commonly adopted approach to runtime verification of memory accesses. However, protection is typically coarse grained where trusted drivers and kernel code is given access to unnecessarily large portions of the memory space. This approach implies huge costs to verification, validation and certification of trusted code.**

In this paper we propose an alternative approach, leveraging on memory safety guarantees of the Rust systems level programming language together with a novel type-based method to statically determine the memory layout of the application. The memory layout information can be used for further static code analysis, proving access pattern properties, and is used as an outset for fine grained hardware supported memory protection.

While the proposed approach is novel and breaks with traditional and industrial practice, our results are applicable to existing hardware platforms and holds the potential to provide leap-step improvements to safety and security of embedded devices.

I. Introduction

The functionality of embedded devices is to an increasing extent software defined. Whereas software based solutions are cost effective and flexible, the problem of ensuring safe and secure operation gets more difficult with device complexity. In particular, increased complexity brings a higher risk of critical bugs while connectivity increases attack surfaces for malicious actors.

At the lowest level, embedded software typically interacts with the underlying hardware through Memory-Mapped Input/Output (MMIO) accesses. A common approach to address safety and security is to leverage on a hardware supported memory protection mechanism in order to isolate untrusted application code from the trusted kernel and driver code. This is manifested by, e.g., ARM TrustZone [1], [2]. The objective is to reduce the amount and complexity of code that needs to be trusted. Still however, the open source OP-TEE TrustZone OS implementation is $>$ 160k lines of C code[3], with a binary size of $>$ 200kB [4]. This coarse grained approach is however heavily flawed as even a single line of code, be it a bug or a deliberate attack, may render the complete system inoperable or even worse, continuing to execute while being exploitable and vulnerable to further attacks with potential collateral consequences. Moreover, adopting a coarse grained approach leads to a large amount of trusted code. In a recent paper [5] Li et. al. survey the TrustZone security architecture and highlight numerous known vulnerabilities. Hadan et al.[6] further elaborate on known vulnerabilities, e.g., "heartbleed" where a memory safety bug affected a majority of OpenSSL based systems and the "goto bug" showcasing that a single line of code is sufficient to break system security.

Traditionally, low level firmware (kernels/drivers etc.) have been implemented in memory unsafe languages such as C and C++. The use of memory unsafe languages is known to directly contribute to a majority of security vulnerabilities[7], and coding standards to reduce risks e.g., MISRA-C[8] are of little to no use [9], even indicating opposite effects.

In this paper we propose an alternative approach, leveraging on memory safety of the Rust systems level programming language (Section II) together with a novel type-based method to statically determine the memory layout of the application (Section III). In short, access to memory regions is tracked through types that dereference at runtime to the underlying physical addresses, which are then used in the usual way for interacting with peripherals through MMIO, or with static heap allocated data for local shared data structures. The memory layout information tracked by the types can then be used for further static code analysis, such as for proving access pattern properties, as well as an outset for fine grained hardware supported memory protection (Section IV). For the former, we can build on readily available methods for symbolic execution of Rust code to validate that all memory accesses adhere to the regions implied by the specified resource types. For the latter case, results are directly applicable to existing Memory Protection Unit (MPU) hardware, where configurations can be automatically generated. Furthermore, one can foresee the generation of custom MPU hardware, which may significantly improve over existing solutions, such a Trust Zone, by offering tamper free memory protection. Finally we conclude the paper and discuss future work (Section V).

II. Background

Rust provides a rich type system with strong guarantees towards memory safety, alias free mutability in particular. However, the memory safety guarantees do not include the explicitly tagged `unsafe` code for which soundness is out of reach for the compiler. Unsafe code is required for dereferencing raw pointers, interaction with external code, e.g., C and inline assembly, and for calling other unsafe code. In the

context of bare metal programming, raw pointers are used to access MMIO and thus underlie all embedded applications.

A. MMIO in Rust

The chip vendor provides documentation for the device along with machine readable metadata. For ARM devices this is typically given in the CMSIS-SVD format, while RISC-V devices usually provide metadata in either CMSIS-SVD or the IP-XACT format. These files contain the peripheral memory mapping and per-register field descriptions. Due to their machine readable XML format they provide an outset for automatic code generation of abstractions of the MMIO registers. The `svd2rust`[10] tool is commonly used to generate a PAC library with a common register-access API, including per-register, type-checked read, modify, and write functions. The abstraction offered by PAC is zero-cost, i.e., the executable code will access the underlying memory without additional overhead. The main drawback is compile time and size of compilation artifacts, as the PACs may consist millions of lines of code for chips with many registers. Other alternatives include `chip-tool`[11], and `stm32ral`[12] which improve on compile time and size of generated artifacts.

At the end of the day, all Rust MMIO leads up to having to dereference the raw pointer, this access being contained in its respective `unsafe`-block. Register access through uniquely borrowed peripheral instances can be considered safe in terms of compliance to Rust aliasing rules. Thus, the generated `svd2rust` API provides safe register read access, while write/modify APIs are safe only for cases where all field values have defined behavior according to the vendor provided XML. This way, the amount of unsafe code required at application level is drastically reduced.[1]

III. Type Based Layout in Rust

Given the outset of MMIO represented by abstractions dereferencing into physical addresses, and that structures overlaying MMIO holds sizing information, we can leverage on the Rust type system as follows.

A. MMIO proxies

`svd2rust` generates Zero-Sized data Types (ZSTs) we refer to as proxies. These proxies dereference to a sized structure overlaying the physical location of corresponding MMIO registers. We define two traits `GetLayoutType` and `GetLayout` for determining the complete layout of a memory object (Listing 1). `heapless`[13] is leveraged to support heap-allocation-free vectors for the layout.

To support the dereference operation, these proxies for peripheral register blocks implement the **`Deref`** trait, automatically generated by `svd2rust`.

We assume types that explicitly implement **`Deref`** to be ZST, which implies that they can be created out of thin air as seen in Listing 2. As `GetLayoutType` is implemented for all `T`, no explicit type bound is required. To obtain the layout of a reference `&T`, we provide the `GetLayout` trait default implementation (Listing 3).

B. Structured Data

To handle structured data in the general case, the layout of each struct field should be added to the layout. This is done through specialization as shown in Listing 4.[2] For the example the data will be allocated on the stack, thus if run on a host the `Layout.address` will vary depending on the stack frame location (and hence the address value cannot be asserted in any meaningful way). In the case of tracking the location of RTIC [14] resources, these will be statically allocated and thus their layout will be stable and thus useful as further discussed in Section IV.

C. Layout Derive

For convenience, a custom derive macro is provided by the `layout-derive`[15] crate. In the `layout-derive` crate repository, further examples are given. Examples include the memory layout for the peripherals of the ESP32-C3.

D. Custom abstractions

While the proposed traits, and implementations thereof, work out of the box for `svd2rust` generated MMIO proxies, special care has to be taken when building further abstractions. This is commonly done by Hardware Abstraction Layer (HAL) crates. For these abstractions to correctly report underlying memory layouts, the custom derive macro needs to be applied to internal data type specifications.

IV. Use of Derived Memory Layout

The derived memory layout can be used for different purposes, either in separation or in combination. In the following we highlight static analysis and run-time verification. Other use cases include automatically generated documentation useful to system certification.

Used in combination with the RTIC declarative task and resource model, we can derive layout information on a per task basis. For non-RTIC use, the relation between tasks and their intended accessible resources must be established by other means, e.g., by some system description/configuration file(s) indicating the set of (typed) resources associated to each task, along with tooling to collect/process the layout information.

[1] `svd2rust` enforces the use of **`unsafe`** code for cases where defined behavior cannot be guaranteed. In Rust, **`unsafe`** typically indicates that memory safety cannot be guaranteed by the compiler. Here, **`unsafe`** implies that defined behavior cannot be guaranteed, a small but significant semantic difference. On one hand, one can argue that undefined behavior at the hardware level violates memory safety per-proxy, while on the other hand, that hardware behavior is out of scope for Rust safety guarantees, thus the adoption of **`unsafe`** is abusing Rust semantics. The problem here is that Rust provides **`unsafe`** as the *only* language level marker for putting additional requirements on calls. A potential solution would be to extend the Rust language with an additional keyword to indicate (potentially) undefined behavior.

[2] As of this writing, **`impl`** specialization requires the nightly toolchain and the `min_specialization` -feature.

Listing 1. Layout Traits

```
1  pub trait GetLayoutType {
2      fn get_layout_type<const N: usize>(layout: &mut heapless::Vec<Layout, N>);
3  }
4
5  impl<T> GetLayoutType for T {
6      default fn get_layout_type<const N: usize>(_layout: &mut heapless::Vec<Layout, N>) {}
7  }
```

Listing 2. Default implementation of GetLayoutType

```
1  impl<T, U> GetLayoutType for T
2  where
3      T: Deref<Target = U>,
4  {
5      default fn get_layout_type<const N: usize>(layout: &mut heapless::Vec<Layout, N>) {
6          // we crate a &ZST out of thin air!!!
7          let t: &T = unsafe { core::mem::transmute(&()) };
8          let data = t.deref();
9          layout.push(Layout {
10             address: data as *const _ as usize,
11             size: core::mem::size_of_val(data),
12         }).unwrap();
13      }
14 }
```

Listing 3. Default implementation of GetLayout

```
1  impl<T> GetLayout for T
2  {
3      default fn get_layout<const N: usize>(&self, layout: &mut heapless::Vec<Layout, N>) {
4          layout.push(Layout {
5              address: self as *const _ as usize,
6              size: core::mem::size_of_val(self.deref()),
7          }).unwrap();
8
9          T::get_layout_type(layout);
10     }
11 }
12
13 impl<T, U> GetLayout for T
14 where
15     T: Deref<Target = U>,
16 {
17     fn get_layout<const N: usize>(&self, layout: &mut heapless::Vec<Layout, N>) {
18         let data = self.deref();
19         layout.push(Layout {
20             address: data as *const _ as usize,
21             size: core::mem::size_of_val(data),
22         }).unwrap();
23     }
24 }
```

A. Static Analysis by Symbolic Execution

Symbolic execution is a verification technique capable of exploring code paths without explicitly enumerating unknown input values. Under the hood, path conditions are modelled in First-Order Logic (FOL), where an unknown value is unconstrained, while a concrete value is bound. The process of symbolic execution boils down to evaluating execution paths under given path constraints. In case of conditionals, new paths are constructed and checked for feasibility typically by solving the corresponding Satisfiability Modulo Theory (SMT) problem. For cases where the SMT problem cannot be solved (due to complexity) approximate solutions can be adopted (essentially reducing constraint complexity by concretization). A prominent example is KLEE[16] with its dynamic execution engine. In prior work, we have shown how Rust code can be symbolically executed using KLEE[17]. In recent work, we have introduced SymEx[18], a symbolic execution engine written in Rust. Unlike KLEE, SymEx constructs and solves each path without approximations.

In Section IV-D, we elaborate on the use of memory layout information for run-time verification of a given RTIC task/resource set. A similar approach can be foreseen, where we use the layout information for static program analysis to assert memory accesses comply to the regions indicated by the task/resource set specification.

We symbolically execute each task, returning fresh unconstrained (symbolic) values on valid MMIO accesses. A failing memory assertion along a feasible path indicates that the

Listing 4. Specialized implementation of `GetLayout` and example of use

```
struct Simple {
    data: u32,
    data2: u64,
}

// this implementation should be generated by a custom derive
impl GetLayout for Simple {
    fn get_layout<const N: usize>(&self, layout: &mut Vec<Layout, N>) {
        // get_layout is executed on each field
        self.data.get_layout(layout);
        self.data2.get_layout(layout);
    }
}

// A usage example may look like this:
fn test_simple() {
    let data = Simple { data: 0, data2: 0 };
    let mut layout: Vec<Layout, 8> = Vec::new();
    data.get_layout(&mut layout);

    assert!(layout[0].size == 4);
    assert!(layout[1].size == 8);
}
```

task attempts to access memory outside of the task/resource specification. The symbolic execution engine provides a concrete assignment of symbolic state leading up to the failing assertion, so the application can be replayed and the bug found. Notice here, with the outset of RTIC based applications, together with the memory safety guarantees provided by the Rust language, memory related violations can only occur in the presence of **`unsafe`** code. To this end, we partially extend the Rust memory guarantees to **`unsafe`** Rust.

B. Run-time Verification for existing MPUs

The RISC-V Privileged Specification[19] defines a memory protection unit – Physical Memory Protection (PMP). An implementation of the PMP is included with the Espressif ESP32-C3 MCU[20]. For the ESP32-C3 implementation the PMP provides 16 unique address registers and corresponding configuration registers defining memory regions and allowed types of access. The PMP is accessible only from privileged machine mode of execution. Once execution enters the non-privileged user mode, the memory protection comes into effect and any disallowed memory accesses cause an exception. The read, write and execute flags of the configuration registers may be combined to define exactly which types of access to a specific region are allowed. The extension allows regions as small as four bytes, meaning the granularity is sufficient to protect memory down to word-wide resources.

To our end, the PMP can be leveraged to protect shared memory by way of generating preludes to user-level tasks at compile time. These preludes configure the PMP to reject accesses outside of the allotted shared resources, letting the task itself run in user mode. To allow the execution of the user task, execute access is allowed in the instruction region of memory.

Accesses to local variables, i.e., data within the memory region allotted to the stack can be filtered using two approaches:

- At run time, a PMP region can be defined between the current task frame pointer and the top of the MCU-wide stack region. Any data in this region is either part of the local call stack, or deallocated.
- The method described in IV-D can be used to determine the task stack depth statically, and the PMP can be configured to allow memory accesses only within that region.

Both methods ensure task separation, whereas the latter approach provides a stronger guarantee for out-of-stack accesses.

In an RTIC context, the granularity of the memory protection can be further improved by letting the resource locking mechanism configure the PMP to allow access to currently held shared resource(s) as further discussed in Section IV-C.

A simple example use case and an implementation, where we allow a user task access to only the GPIO peripheral using the automatically derived layout can be found at the `esp32c3-pmp` GitHub repository[21]

C. Custom Hardware Generation for Run-time Verification

The open source nature of the RISC-V architecture allows for custom extensions to the architecture. Using modern FPGA technology, custom architectures can be rapidly prototyped. FPGA implementations may in many cases suffice for use in end products, with advantages garnered in time-to-market, flexibility, low-cost in short series, etc. For larger quantities and/or higher performance, a custom ASIC might be preferable where FPGA prototyping is typically adopted in preceding steps.

The presented per task memory layout can be used to generate HDL descriptions for application-specific MPU configuration, where the number of MPU regions can be set to match exactly the task set requirement. For RTIC based systems, tasks are mapped 1:1 to interrupts. The generated MPU can leverage on that property and automatically switch to the corresponding configuration.

As an effect, both configuration and configuration selection can be performed directly in hardware. This brings numerous advantages:

- Zero latency MPU configuration switching on enter/exit of interrupts, and
- memory protection guarantees robust to software bugs and tampering.

Using RTIC, we can even go further regarding granularity of memory accesses. As the use of a shared resource (e.g., an MMIO peripheral) implies the locking operation of the corresponding resource, we can statically determine a mapping between each held lock and the resource memory layout behind the lock. Thus, fine grained (per-task/per-resource) grants can be achieved with minimal software involvement.

D. Custom MPU example

Let us assume a system with three tasks t_1, t_2, t_3, bound to interrupts i_1, i_2 and i_3 respectively. The system has 4 resources, $r_1 \ldots r_4$, with a task/resource grants as per Table I, where $r1$ is a local resource (an `[u8;8]` array), $r2$ and $r3$ MMIO peripherals and $r4$ a shared resource (a `ComplexStruct` structure) with layout as per Table II.

TABLE I
TASK/RESOURCES ACCESS GRANTS.

Task	Resources
t_1	r_2, r_4
t_2	r_1, r_3, r_4
t_3	r_2, r_3

TABLE II
RESOURCE LAYOUT

Resource	Address	Size
r_1	0x2000_0020	0x8
r_2	0xc001_0100	0x20
r_3	0xc020_4000	0x10
r_4	0x2000_1200	0x400

Custom MPU generation is straightforward. Address validation starts with the knowledge of actively running interrupt. E.g., i_2 (t_2, due to the 1-1 task/interrupt mapping). For t_2, the MPU holds a bit-vector indicating $r_1|r_3|r_4$ (allowed accesses should be in corresponding ranges). This check can be implemented by parallel hardware, one address comparator for each resource in the system, thus a logic or ($\vee$) between active comparators should yield true for any valid memory operation.

This design holds true for all resource memory accesses, but what about local stack accesses?

To this end, we can leverage on `cargo-call-stack`[22], to statically determine the stack depth of each task. The tool was developed as part of a Master's thesis at LTU[23]. At interrupt entry, the s field should be updated based on the current stack pointer `sp` core register value - the stack memory requirement for the corresponding interrupt/task. In Table III we show the state of the MPU table where we have entered interrupt i_2 with an initial `sp 0x2000_0f84`. For a pure hardware implementation, tight coupling to the CPU core is required due to the access of the `sp` register.[3]

In future work, we project to prototype the implementation of custom MPUs including the aforementioned per lock granular approach.

TABLE III
MPU LAYOUT

Resource	Address	Size
s	0x2000_0f64	0x20
r_1	0x2000_0020	0x8
r_2	0xc001_0100	0x20
r_3	0xc020_4000	0x10
r_4	0x2000_1200	0x400

E. Custom MPU architecture

Figure 1 illustrates a minimal custom MPU architecture. Inputs to the MPU are the address to check (`PC/DM address`) and the current task identifier (`Current Task Id`). There is a single output: `illegal memory access`. The `Tid Region Mapping` component implements the task/resource grants (e.g., Table I). The resource layouts (e.g., Table III) are matched in parallel (`Region 0..N-1` and `Stack Protect` components).

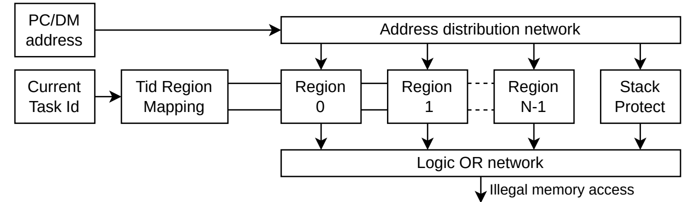

Fig. 1. Custom MPU architecture.

The design is fairly straightforward. High fan-out/fan-in problems of the address distribution network (input) and logic or network (output) may be implemented by tree architectures (with logarithmic critical path complexity). The memory requirement in `Tid Region Mapping` is $T * R$ bits, where T is the (max) number of tasks, and R the (max) number of resources. The `Stack Protect` will have a memory requirement of $T * W$ where T is the number of tasks and W is the bit-with of the max stack region associated to any task. On dispatch of task T_i from task T_j the SP region is reduced by $W(T_j)$, thus protecting T_j from being accessed by T_i. (This holds recursively, thus T_i cannot access any task on the stack besides its own region.)[4]

The minimal design can be straightforwardly extended to discriminate per task, per resource access rights. While the design would remain largely the same, the required memory in

[3]As RTIC unifies tasks and interrupts, all tasks runs on a shared stack, thus the allowed stack region will change dynamically. Non-RTIC implementations typically adopt (wasteful) static allocation of stack regions for each task, so the mechanism to determine stack base address and size would differ slightly.

[4]This approach is sufficient to ensure stack isolation between tasks, but not isolation on a call by call bases within each task.

the `Tid Region Mapping` component would double (using individual bits for *read* and *write* access, where *read/write* access amounts to both bits set).

Table IV shows the results of a hand written implementation of the custom MPU architecture for the introduced example onto a Xilinx `xcvu9pflga2104-2L`. As seen the MPU is synthesized into a fully combinational circuit (zero cycle latency) occupying < 0.01 of the available CLB/LUT resources. While the example demonstrates the feasibility of the approach, it is too small to evaluate realistic cost and performance.

TABLE IV
MPU SYNTHESIS

Site Typ e	Used	Fixed	Available	Util %
CLB LUTs	31	0	1182240	< 0.01
LUT as Logic	31	0	1182240	< 0.01
LUT as Memory	0	0	591840	0.00
CLB Registers	0	0	2364480	0.00
Register as Flip Flop	0	0	2364480	0.00
Register as Latch	0	0	2364480	0.00
CARRY8	0	0	147780	0.00
F7 Muxes	0	0	591120	0.00
F8 Muxes	0	0	295560	0.00
F9 Muxes	0	0	147780	0.00

V. CONCLUSIONS AND FUTURE WORK

We have reported on a novel type-based approach to statically determine the memory layout of applications written in the Rust language. It provides the outset for fine granularity memory protection with precision far superior to traditional coarse grained mechanisms (e.g., TrustZone based solutions).

In the paper we have covered the mechanisms behind type based layout, and discussed potential use of results to both static analysis and run-time verification by vanilla MPUs as well as custom generated MPU hardware.

While static analysis ensures the validity of the application to its specification (in terms of per task accessible resources) run-time verification brings additional resilience to toolchain bugs (compiler/linker faults). Additionally, custom generated MPUs provide hardening to run-time tampering.

Our early experiments indicate the feasibility of low/zero clock cycle latency custom MPU implementations onto commonplace FPGAs.

In future work we project to further explore the use of results to static analysis, vanilla MPUs (e.g., the RISC-V PMP) and custom MPU hardware generation. Another topic of interest is to extend the model to distinguish between different access types (read, write and read/write). Further work can be envisioned towards validating peripheral Direct Memory Access (DMA). The working hypothesis here is that a DMA transactions can be seen as tasks from a modelling perspective. These tasks are run-to-end in compliance to the SRP[24] requirement underlying the RTIC scheduling model. Moreover, DMA tasks are comparably simple as they assume all resources are owned in Rust terms, thus lock free.

REFERENCES

[1] "TrustZone for Cortex-A." [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-a

[2] "TrustZone for Cortex-M." [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-m

[3] Synopsys, "OP-TEE OS analysis scan, 2021," 2023. [Online]. Available: https://scan.coverity.com/projects/op-tee-optee_os

[4] TrustedFirmware.org., "OP-TEE OS Frequently Asked Questions," 2023. [Online]. Available: https://optee.readthedocs.io/en/latest/faq/faq.html?highlight=size#q-what-is-the-size-of-op-tee-itself

[5] W. Li, Y. Xia, and H. Chen, "Research on arm trustzone," *GetMobile: Mobile Comp. and Comm.*, vol. 22, no. 3, p. 17–22, jan 2019. [Online]. Available: https://doi.org/10.1145/3308755.3308761

[6] H. Hadan, N. Serrano, and L. J. Camp, "A holistic analysis of web-based public key infrastructure failures: comparing experts' perceptions and real-world incidents," *Journal of Cybersecurity*, vol. 7, no. 1, p. tyab025, 12 2021. [Online]. Available: https://doi.org/10.1093/cybsec/tyab025

[7] NSA - National Security Agency, "Software Memory Safety," 2023. [Online]. Available: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[8] "MISRA." [Online]. Available: https://misra.org.uk/

[9] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *2008 IEEE International Conference on Software Maintenance*, 2008, pp. 277–286.

[10] Embedded Rust Tools Team, "svd2rust - Generate Rust register maps from SVD files," 2023. [Online]. Available: https://github.com/rust-embedded/svd2rust

[11] Dario Nieuwenhuis & contributors, "chiptool - chiptool is an experimental fork of svd2rust," 2023. [Online]. Available: https://github.com/embassy-rs/chiptool

[12] Adam Greig & contributors, "stm32ral - Rust RAL (register access layer) for all STM32 microcontrollers," 2023. [Online]. Available: https://github.com/adamgreig/stm32ral

[13] Jorge Aparicio & contributors, "heapless - Heapless, 'static' friendly data structures in Rust," 2023. [Online]. Available: https://github.com/japaric/heapless

[14] RTIC Team, "The RTIC Book," 2023. [Online]. Available: https://github.com/rtic-rs/

[15] Per Lindgren, "layout-derive," 2023. [Online]. Available: https://github.com/perlindgren/layout-trait

[16] Cristian Cadar, Daniel Dunbar, Dawson Engler & contributors, "KLEE - A Dynamic Symbolic Execution Engine," 2023. [Online]. Available: http://klee.github.io/

[17] M. Lindner, J. Aparicius, and P. Lindgren, "No panic! verification of rust programs by symbolic execution," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, 2018, pp. 108–114.

[18] J. Norlen, "Architecture for a Symbolic Execution Environment," 2022. [Online]. Available: http://ltu.diva-portal.org/smash/get/diva2:1688170/FULLTEXT01.pdf

[19] "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture," 12 2021, accessed 11.3.2023. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[20] Espressif, "ESP32C3 Technical Reference Manual," 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf

[21] Pawel Dzialo, "esp32c3-pmp," 2023. [Online]. Available: https://github.com/onsdagens/esp32c3-pmp

[22] Jorge Aparicio, "cargo-call-stack," 2023. [Online]. Available: https://github.com/japaric/cargo-call-stack

[23] J. Aparicio Rivera, "Real time rust on multi-core microcontrollers," Master's thesis, Luleå University of Technology, Computer Science, 2020.

[24] T. P. Baker, "A stack-based resource allocation policy for realtime processes," *[1990] Proceedings 11th Real-Time Systems Symposium*, pp. 191–200, 1990.