Minimally Intrusive Safety and Security Verification of Rust RTIC Applications

Pawel Dzialo*, Ivar Jönsson†*, Malte Münch*, Erik Serrander†, Johan Eriksson†, Per Lindgren*†
{pawel.dzialo, malte.munch, per.lindgren}@ltu.se
*Luleå University of Technology
{ivar.jonsson, erik.serrander, johan}@grepit.se
†Grepit AB

***Abstract*—In the context of formal methods, symbolic execution stands out as highly automatic, allowing static code analysis to be adopted by users without domain specific expertise or training.**

Symex is a symbolic execution framework specifically targeting analysis of embedded software with requirements to safety, security, and dependability. However, so far, the execution engine of Symex has been executing the LLVM Intermediate Representation(LLVM-IR). Because no consistent mapping between LLVM-IR and actual machine code exists, Symex has been unable to provide guarantees to runtime properties of the system, such as Worst-Case Execution Time (WCET).

In this paper, we extend Symex by moving the execution engine to *General Assembly* (GA), an Intermediate Representation (IR) capable of capturing the semantics of Instruction Set Architectures (ISAs), along with their non-functional properties, e.g. per-instruction execution time, or power consumption. Symex lifts the ELF binary to GA and explores all reachable paths without approximations, thus it is able to provide guarantees to runtime characteristics of the system, taking into account architecture specific behaviour and compiler backend/linker optimizations.

Furthermore, we introduce the EASY system analysis framework, which uses Symex as a symbolic execution backend, and therefore grants the same verification capabilities. EASY can provide analysis for response time, task memory isolation and application stack memory utilization.

We demonstrate the feasibility of GA-based symbolic execution by modelling the full ARMv6 and most of the v7 ISA, as well as the RV32I ISA. Leveraging on the Rust RTIC framework, we demonstrate that EASY is capable of automatically determining the schedulability, worst-case stack memory utilization and task memory isolation properties of the system.

***Index Terms*—Symbolic Execution, Formal Verification, WCET, Response time analysis, RTIC, Memory-Safety, Automatic, Non-intrusive**

I. Introduction

Embedded software plays an increasingly important role to the functionality and operation of embedded systems, thus dependability relies to a high degree on the performance, robustness, reliability, safety and security of the firmware, calling for rigorous firmware validation. Further adding to the complexity of validation, embedded software typically operates under non-functional requirements and constraints (timing, memory footprint and usage, power, low-level hardware interaction, etc.).

Whereas formal methods may offer stronger confidence and trust, wide adoption is hindered by high instep cost (requiring skills/training) and adds complexity and increased effort to the verification process in comparison to commonplace test based approaches.

In this paper we present progress on Symex [1] and EASY [2]. Symex is an open source framework for symbolic code execution aiming to lower the instep cost and efforts typically associated with the use of formal methods for static analysis of embedded software. While symbolic execution is a well established technique, existing tools such as KLEE [3] rely on *dynamic* execution engines (approximating constraints)[1]. Symex, instead adopts a *pure* approach (where path constraints are free of approximation) and is thus useful for automatically obtaining *guarantees* about the runtime behavior of the system[2]. In this paper, we extend the Symex execution engine to operate on *General Assembly*, an Intermediate Representation (IR) capable of capturing the semantics of commonplace Instruction Set Architectures (ISAs) along with their non-functional properties (such as per-instruction execution time, power consumption, etc.). Moreover, the framework is extendable, where target and platform specific behaviour can be modelled by means of closures (functions operating on simulation state). Together, this allows non-functional specifications to be captured, and applications to be verified and proven against both functional and non-functional requirements.

The EASY framework uses Symex to provide automatic full-application verification against a user provided specification expressed as an (extended) Rust RTIC model. In this paper we show that EASY is capable of verifying: per-task response time, per-task memory isolation and safe bounds to worst case application stack-memory utilization. Furthermore, additional verification conditions can be added via assertions.

Key contributions in summary:

- Extending the Symex symbolic execution framework:
  - Section III-A, General Assembly extension, allowing binary level code analysis.
- The EASY framework, implementing a novel, symbolic execution-based methodology for system wide safety and security validation of specifically Rust RTIC applications:
  - Section IV-B1, safe (upper bound) WCET for tasks and inner resource locks (critical sections).
  - Section IV-B, Stack Resource Policy (SRP) based scheduling analysis.
  - Section IV-C, safe upper bound worst case stack memory estimation.
  - Section IV-D, model based per-task memory access analysis for static task isolation guarantees.

This research is financially supported by the European Regional Development Fund and the Cybersäkerhetsnod Norr-project (no. 20366918) and the Swedish Innovation Agency 2025-00844, CRIT - Certification of RUST in crITical Systems.

[1]Approximation is a deliberate choice taken allowing progression in case the underlying Satisfiability Modulo Theories(SMT) solver fails within given time/memory limits to come up with a result.

[2]We treat models for which the generated path constraints are unsolvable within given memory and time constraints by the underlying SMT solver as erroneous.

II. Background

A. Rust

The Rust programming language [4] is a system-level programming language providing compile-time guarantees to the memory safety, and freedom from undefined behavior of the application.

By design, in cases where the problem is inherently unsafe, these guarantees can be circumvented by marking a code block as explicitly `unsafe`. For instance, in an embedded context, access to Memory Mapped Input/Output (MMIO)-based peripherals requires raw pointer dereferencing, which is, by definition, unsafe, and the root cause for a whole class of memory-related vulnerabilities [5]. This is typically addressed by wrapping direct peripheral register access in safe abstractions on the Hardware Abstraction Layer (HAL) level [6]. However, the underlying HAL implementation still builds on `unsafe` code, and relies on manual verification to ensure safety.

B. Stack Resource Policy

In [7], an extension to the Priority Ceiling Protocol synchronization protocol named the Stack Resource Policy (SRP) is introduced. In the context of this paper, the underlying formal model can be defined as follows:

- τ_i A task τ_i is a finite (terminating) sequence of instructions to be executed on a single processor core.
- P_i The (base) priority of τ_i. If $P_a > P_b$, the execution of τ_a is of higher importance than τ_b. As such, the execution of τ_b is delayed.
- π_i The preemption level of τ_i. τ_a may only preempt τ_b if $\pi_a > \pi_b$.
- S_k A shared system resource.
- $L(S_k)$ The set of tasks that may request S_k.
- $\lceil S_k \rceil$ The priority ceiling of S_k, defined as:

$$\lceil S_k \rceil = \max(\{0\} \cup \{\pi_i | \tau_i \in L(S_k)\}) \tag{1}$$

- $Z_{i,k}$ During the execution of critical section $Z_{i,k}$, τ_i may access S_k.
- Π The current (dynamic) system ceiling.

Under SRP, τ_i is only dispatched for execution if P_i is the highest among all outstanding task requests, and $\pi_i > \Pi$.

1) Properties: In [7], a set of properties of systems scheduled using SRP is derived: race- and deadlock freedom, bounded priority inversion, single context switch per task request, single (shared) execution stack, and amenability to static response time and schedulability analysis. These provide a sound outset for designing robust and efficient hard real-time systems.

C. RTIC

Real-Time Interrupt Driven Concurrency (RTIC) [8] is a Rust framework building on SRP. RTIC provides Rust macros, using which the programmer can define a compileable SRP task/resource model. Examples of RTIC task syntax are displayed in Listings 3 and 4.

Traditionally, the preemption level of a task is set dynamically, allowing e.g. Earliest Deadline First (EDF) [9] scheduling. The original work on RTIC [10] instead restricts SRP to static preemption levels, i.e. $\pi_i = P_i$. This allows RTIC to leverage on commonplace interrupt controllers for scheduling acceleration, as shown in [10] [11], and [12]. In [10], SRP is additionally restricted to single-unit resources, defining the computation of Π as follows:

$$\Pi = \max(\{0\} \cup \{P_c\} \cup \{\lceil S \rceil | S \in S_c\}) \tag{2}$$

where P_c is the priority of the currently executing task, and S_c is the set of currently claimed resources. This allows efficient (constant time, on most supported architectures single-cycle) management of Π.

The formal underpinning means any compiling RTIC application comes with the guarantees provided by SRP. Additionally, the Rust programming language provides further guarantees to memory safety, and defined behavior, providing a solid basis for implementing safe and secure applications. Notice that the limitations mentioned in Section II-A apply, meaning that in some cases, the Rust language guarantees regarding memory safety are insufficient to guarantee total memory safety.

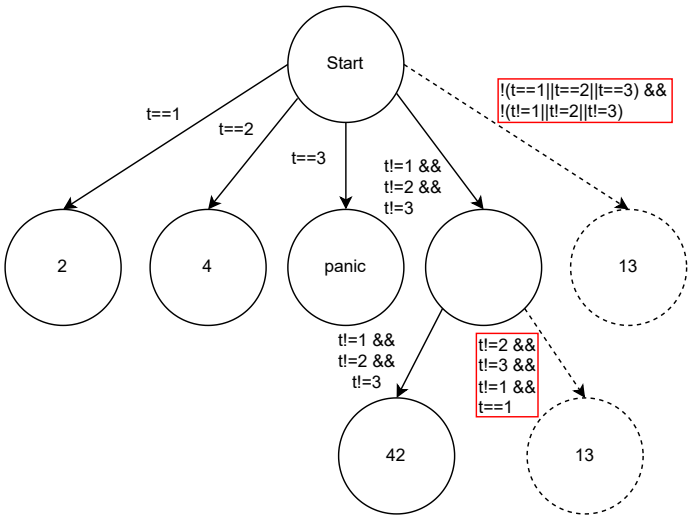

Fig. 1: Symbolic execution search tree according to Listing 1. Generated path conditions are shown on edges.

D. Symbolic Execution

Symbolic execution is an automatic formal verification technique assigning each input (or state) an initially unconstrained (or assumed) symbolic value, and performing an exhaustive search over all of the possible code paths. An execution engine keeps track of the symbolic variables, and strengthens their constraints according to path conditions. Path feasibility is determined by solving corresponding Satisfiability Modulo Theories (SMT) [13][3] problems, which allows unfeasible paths to be excluded from further exploration. Using the simple function in Listing 1 as an example, the corresponding search tree is displayed in Figure 1, with symbolic variable values displayed as edge labels, and unfeasible paths denoted as dashed nodes/edges.

E. Symex

Symex is a symbolic execution engine originally operating on the LLVM Intermediate Representation(LLVM-IR) [14], which is a processor architecture agnostic intermediate representation, typically used as input to the the LLVM compiler infrastructure [15].

However, due to the lack of a defined mapping between LLVM-IR and machine code it is difficult to draw definitive conclusions about run-time characteristics (specifically Worst-Case Execution time (WCET)) of the resulting binary [16]. Additionally, any verification made on LLVM-IR level must also assume that the compilation of LLVM-IR to machine code is correct in all cases.

[3]Further explained in Section II-E1

```rust
fn simple(t: u32) -> u32 {
  if t == 1 {
    return 2;        // path 1
  } else if t == 2 {
    return 4;        // path 2
  } else if t == 3 {
    panic!()         // path 3
  } else {
    if t == 1 {
        return 13;   // unreachable
    }
    return 42;       // path 4
  }
  return 13;        // unreachable
}
```

Listing 1: The simple Rust function under analysis.

1) SMT Solvers: SMT solvers solve the boolean satisfiability problem with respect to a set of theories such as the theory of fixed size bitvectors, a theory central to Symex. SMT solvers typically support all of or a subset of the SMT-LIB standard [17] which serves as both a standard for benchmarking and a generic interface language for solvers. Symex operates using SMT solvers as a backend to determine if paths are feasible during analysis.

At the time of writing, the Symex framework supports two solvers, namely Boolector and Bitwuzla.

Boolector [18] is a fast SMT solver with support for all of the SMT-LIB theories [17] required by Symex. However, it lacks support for the theory of floating point arithmetic. Supporting this theory is not a requirement, however, it would allow for tighter analysis results for code involving floating point values.

To address this issue, recent work on Symex includes supporting the Bitwuzla solver. Bitwuzla [19] is the successor to Boolector, introducing improvements and additional features, including support for floating point arithmetics.

F. Related Work

1) aiT: aiT [20] is a schedulability analysis tool that provides WCET estimation and can perform response time analysis using RT-Druid [21].

As outlined in [20], aiT uses control flow graphs to determine possible paths through the program. While this method allows for binary analysis in a similar fashion as in Symex, it may limit the precision in programs that use recursive function calls. aiT solves this by introducing loop bounds externally, in contrast Symex and EASY do not need externally defined loop bounds as the executor emulates the execution of the firmware binary.

2) BinSEC: BINSEC is a collection of security-focused binary-level analysis and testing tools, which includes a symbolic execution engine capable of performing static-, dynamic- and relational symbolic execution [22]. Some of the available tools, plugins and extensions in the framework are highly relevant to the work done on Symex, notably its binary-level dynamic symbolic engine, BINSEC/SE [23]. BINSEC/SE relies on execution traces to guide path exploration, improving scalability at the cost of limiting its ability to provide formal correctness proofs. Symex, instead uses pure symbolic execution to explore all feasible execution paths.

Recent work on BINSEC/CODEX [24] introduces binary level static analysis for automated verification of C-programs (specifically embedded OS kernels), proving absence of privilege escalation and runtime errors - essentially verifying memory safety for C code.

III. Symex: Binary Level Execution Engine

A. Symex

To address the limitations of Symex described in Section II-E, we have modified the tool to work directly on the compiled ELF binary. This is done by lifting the architecture specific machine code to *General Assembly* (GA) on which the new execution engine operates. The GA language captures the ISA semantics along with non-functional properties (allowing to model cycle accurate execution time, power characteristics etc.). As the GA backend is not tied to any specific processor architecture, it is sufficient to provide the mapping between machine code and GA for the architecture at hand. By operating at binary level, the tool is input language agnostic and able to handle both inline assembly and externally linked code. Currently, Symex fully models the ARMv6-M ISA, while partial modelling of ARMv7-EM and experimental RISC-V (RV32I) modelling is under evaluation. More information on GA, omitted from this paper due to space constraints can be found in the associated master's thesis [1].

IV. The EASY Methodology

To demonstrate our proposed analysis methodology, we introduce three code examples, presented in Listings 1, 3 and 4. The code in Listing 1 depicts a minimal example of a function, demonstrating the principles of symbolic execution. The code in Listings 3, 4 shows a more complete example of an RTIC application, in this case a naive UART echo, which is the focus of the rest of this section.

A. RTIC application

At its core, an RTIC application is a declarative SRP model consisting of task and resource declarations. Example task declarations are displayed in Listings 3 and 4. The `#[task(...)]` attribute macro declares task characteristics: the bound interrupt vector (`binds = ...`), the set of resources local (exclusive) to the task (`local = [...]`), the set of resources shared with other tasks (`shared = [...]`), and the priority of the task (`priority = ...`). The `#[task(...)]` attribute presented in this paper contains fields not found in upstream RTIC applications (`period`, `deadline`, `frequency`) . We will come back to those in the following subsection.

B. Schedulability analysis

RTIC-based applications adhere to static priority SRP-based scheduling. As such, the worst-case response time R_i of each task τ_i can be obtained by the recurrence relation outlined in [25]:

$$\begin{cases} R_i^0 = C_i + B_i \\ R_i^s = C_i + B_i + \sum_{h:P_h>P_i} \lceil \frac{R_i^{(s-1)}}{T_h} \rceil C_h). \end{cases} \tag{3}$$

where C_i, P_i, T_i, and B_i are the WCET, priority, minimum inter-arrival time, and blocking time of τ_i respectively. Obtaining the blocking time B_i is also described in [25], as first constructing the set γ_i of critical sections $Z_{j,k}$ belonging to task τ_j, over resource S_k with the priority ceiling $\lceil S_k \rceil$, such that

$$\gamma_i = \{Z_{j,k} | P_j < P_i \wedge P_i < \lceil S_k \rceil)\} \tag{4}$$

From here, the blocking time can be calculated by

$$B_i = \max_{j,k} \{\delta_{j,k} - 1 | Z_{j,k} \in \gamma_i\} \tag{5}$$

where $\delta_{j,k}$ is the WCET of the lock over S_k belonging to τ_j.

Intuitively, if and only if for all tasks τ_n, the deadline $D_n \geq R_n$, the system can be deemed schedulable [25].

```
-#[rtic::app(
+#[easy::rtic::app(
+  frequency=64MHz,
)]
mod app {
  #[task(
+    deadline = 5us,
+    period = 10us,
// or
+    frequency = 100kHz
    }
  )]
  fn task...
}
```

Listing 2: The syntax changes to an RTIC application needed to permit full application schedulability analysis. The added lines are marked with green, while the removed lines are marked with red.

To allow automatic schedulability analysis of RTIC applications, we start by extending the RTIC syntax with the model parameters of the above equations, D_n and T_n (the `deadline`, `period`, task attribute macro fields already mentioned in Section IV-A). Alternatively, the task period may also be expressed as an arrival frequency through the `frequency` field[4]. If no explicit deadline is specified, it is assumed to be equal to the minimum inter-arrival time of the task, i.e. the most relaxed deadline yielding a schedulable system with one-sized event buffers, a common assumption in the context of hard real-time systems [25]. Additionally, to allow the expression of periods and deadlines in terms of wall-clock time instead of core clock cycles, the overall application declaration now includes the configured clock frequency in the application wide `frequency` attribute. The changes introduced to the RTIC syntax are displayed in Listing 2

1) Worst-case execution time estimation: Owing to the structure of an ELF binary, the WCET C_n of any task τ_n can be obtained by looking up the corresponding symbol i.e. function entry point, bounding the function through the architecture-specific return instruction, and performing analysis on the obtained machine code block using Symex.

Obtaining the WCET of critical sections is a more complicated matter because there exists no explicit debug symbol marking the entry/exit of a resource lock. Here, we exploit the fact that RTIC locks are typically[5] implemented through global interrupt mask (e.g. Cortex-M Nested Vector Interrupt Controller(NVIC) Interrupt Set Enable Register(ISER)) or priority threshold (e.g. NVIC BASEPRI and PRIMASK) manipulation. By hooking writes to these configuration registers, we can detect the entry/exit points of resource locks and bound the critical code section accordingly. From here, the WCET of the critical section can be obtained in the same manner as for entire tasks.

2) Platform specific hardware models: In some cases, hardware interaction in embedded firmware requires the core to block execution (busy-wait) until a given event occurs. It is possible to provide accurate timing analysis for such code. Symex does not hard-code timing characteristics for instructions, instead, these can be supplied by arbitrary hardware models. Using this approach, it is possible to model a wide variety of actual hardware.

[4]Not to be confused with the application-wide `frequency` attribute, denoting the core frequency

[5]For all currently supported architectures.

Modeling hardware requires knowledge of the specific System-on-Chip (SoC), making them platform specific. At the time of writing, partial support for the Microchip SAM-{E, S, V}7 platform [26] has been implemented. The supported peripherals include temporal models of the TWI, UART and RTT peripherals, allowing analysis of blocking interactions. Additionally, simpler models have been implemented for MCAN, TC, SPI, SSC, HSMCI, PWM, and USART. These simply return an unconstrained symbolic value whenever read, thereby always over-approximating the number of paths such an interaction can yield.

C. Stack memory usage

The worst-case stack memory usage of a task can be determined in a manner similar to its worst-case execution time, by tracking the code path yielding the lowest concrete value of the stack pointer (as stack grows towards lower addresses). This information is directly useful to determining safety properties of the system, by verifying that the worst-case stack usage does not exceed the size of the block of memory allotted to the stack.

D. Task isolation analysis

Under RTIC, the memory the task is expected to access can be divided into two categories: statically allocated resources (shared and local), and the local stack. Because of static allocation of resources, information about the size and location of resources is known at compile-time.

Since the task-resource mapping is a direct model parameter, the specific resources to which a task is intended to have access can be used to derive the layout of the memory regions to which a task is allowed access. Additionally, as the expected stack memory usage for a given code path is known, this information can be used to verify each memory access made. By treating any accesses outside of the declared allowed regions (i.e. resources + local stack) as a violation, we can additionally ensure complete isolation between tasks, even in presence of explicitly `unsafe` code blocks.

E. Requirement adherence analysis

For all code besides explicitly marked `unsafe` blocks, the Rust compiler automatically generates assertions that call `panic` if violated. Thus, as proposed in [27], by proving all paths to `panic` to be unreachable, defined behavior can be ensured at compile time. We use a similar approach, by treating the correctness of the system as absence of `panic`. Through Rust `assertions`, the end-user can, programatically, and according to system-level requirements, introduce additional bounds on the behavior definition under analysis (e.g. constraints on the value of some interface).

Assertions can potentially incur performance overhead, discussed further in Section VI-A7.

V. Results

In this section, we apply the methods described in Section IV to the code examples displayed in Listings 1, 3 and 4

A. Execution time estimation

We start by looking at the simple example displayed in Listing 1. The results of compiling the function for the three targets supported by Symex (ARM v6m, v7em and RV32I) are displayed in Figure 5. To the reader familiar with RISC-V and ARM assembly, the execution time of the function may be obvious already at this point. Regardless, we perform execution analysis of the function using Symex, and compare the results to actual measurements made on the nRF52840 Cortex-M4 (v7em), and Hippomenes (single-stage RV32I synthesized

to FPGA). As shown in Tables I and II, the Symex-generated worst-case paths, and their calculated execution times are safe estimates in comparison to the measurements.

```
00000fa8 <simple>:
cmp     r0, #0x1
itt     eq
moveq   r0, #0x2
bxeq    lr
cmp     r0, #0x2
itt     eq
moveq   r0, #0x4
bxeq    lr
cmp     r0, #0x3
itt     ne
movne   r0, #0x2a
bxne    lr
push    {r7, lr}
mov     r7, sp
bl
0xfc8 <panic>
```

Fig. 2: v7em

```
00001066 <simple>:
push    {r7, lr}
add     r7, sp, #0x0
cmp     r0, #0x1
beq
0x1076 <+0x10>
cmp     r0, #0x2
bne
0x107a <+0x14>
movs    r0,#0x4
pop     {r7, pc}
movs    r0, #0x2
pop     {r7, pc}
cmp     r0, #0x3
beq
0x1082 <+0x1c>
movs    r0, #0x2a
pop     {r7, pc}
bl 0x1086 <panic>
```

Fig. 3: v6m

```
00000000 <simple>:
li      a1, 0x1
beq     a0, a1, 0x18
li      a1, 0x2
bne     a0, a1, 0x20
li      a0, 0x4
ret
li      a0, 0x2
ret
li      a1, 0x3
beq     a0, a1, 0x30
li      a0, 0x2a
ret
auipc   ra, 0x0
jalr    8(ra)<panic>
```

Fig. 4: RV32I

Fig. 5: The results of compiling Listing 1 for the three target architectures supported by Symex.

Path ID	Return Value	Symex path estimation (cycles)		
		v7em	v6m	rv32i
1	2	6	13	4
2	4	11	14	6
3	panic	23	15	8
4	42	16	17	8

TABLE I: Lists Symex execution time estimates for the example code in Listing 1. Estimation bounds are deduced from the publicly available documentation [28], tighter results for the ARM architectures require hardware specification (not publically available). Notice, Symex successfully proves the unreachability of paths leading to `return 13;`.

Path ID	Return Value	Measured execution (cycles)	
		v7em	rv32i
1	2	13-9	4
2	4	19-9	6
3	panic	(A) -	(B) 8
4	42	24-9	8

TABLE II: Hardware measurements for the example code in Listing 1. For the v7em, measurements include the overhead of a function call from (and return to) instrumentation code (measured to 9 clock cycles). Adjusted measurements show that Symex provides safe clock cycle estimates. For (A) the panic handler does not return to the instrumentation code, thus no measurement is obtained. (B) reports cycle time until the panic handler is reached.

B. Buffer overflow

Listing 3 shows a simple UART driver. The driver receives bytes and writes them to a shared buffer `data`. `data` is intended to contain the last 4 bytes read, however, in the example, we've included an incorrect bound on the array index. Because of this, the task could potentially write outside of the allocated array. Performing analysis according to Section IV on this task correctly identifies that the buffer can be exceeded, i.e. that a write may occur outside of the memory regions allotted to the task, resulting in an assertion violation (error). If the bound to reset `idx` is changed to when it exceeds the size of the array, the code in Listing 3 has no paths that violate the task isolation property, and the analysis passes.

```
#[task(
  binds = UART0,
  local = [rx,idx:usize = 0],
  shared = [data],
  spawns = [tx],
  priority = 3,
  frequency = 14400hz, // 115200 / 8
)]
#[inline(never)]
fn rx(mut cx: rx::Context) {
  let rx = cx.local.rx;
  rx.clear_interrupt();
  if let Ok(message) = rx.read() {
    if *cx.local.idx > 4 {
      *cx.local.idx = 0;
      tx::spawn().unwrap();
    }
    cx.shared.data.lock(|data| unsafe {
      *data.get_unchecked_mut(*cx.local.idx) =
        message
    });
    *cx.local.idx += 1;
  }
}
```

Listing 3: A simple UART receiver implementation with a buffer overflow vulnerability.

```
#[task(
  priority = 1,
  local = [tx],
  shared = [data],
  frequency = 3600hz, // 14400/4
)]
#[inline(never)]
fn tx(mut cx: tx::Context) {
  let bytes = cx.shared.data.lock(
    |data| data.clone()
  );
  for byte in bytes {
    while let Err(nb::Error::WouldBlock) =
      cx.local.tx.write(byte) {}
  }
}
```

Listing 4: A simple UART transmit implementation that transmits four bytes of data at a time in a blocking manner.

C. Schedulability analysis

By combining the `rx` task in Listing 3 with a transmit task that iteratively transmits the bytes in the buffer back to the host as shown in Listing 4, we form a complete RTIC application. We perform analysis of the resulting application, and present the results in Tables III and IV.

VI. Discussion

The implementation of the methodology presented in this paper is collected under a tool named Execution Analysis using SYmbolic execution (EASY), described further in a master's thesis [2], which includes results of applying the method to more realistic examples, omitted from this paper due to space constraints. Although the

Task	rx	tx
WCET[μs]	8.1	216.4
Time blocked[ns]	708.3 by tx	N/A
Time preempted[μs]	0	32.5
Response time[μs]	8.8	248.9

TABLE III: EASY generated SRP analysis for a simple RTIC application composed of the tasks in Listings 3, and 4 assuming a core-frequency of 240MHz, the UART running at 115200Baud and that the core is running from wait state free memory.

Task	rx	tx	App
Deadline[μs]	69.4	277.8	N/A
Core utilization	11.7%	77.9%	89.6%
Schedulable	Yes	Yes	Yes
Stack usage[bytes]	104	72	176

TABLE IV: EASY generated SRP analysis results based on the results in Table III. The row with Task name App refers to the entire applications utilization to show that the entire application is schedulable.

presented results confirm the claims presented in Section IV, a number of limitations apply.

A. Limitations

1) ARM-based targets: The performance of an ARM core is affected by the underlying implementation of the branch predictor and pipeline stages, the models of which are unavailable to the public. Without this information, deriving accurate figures regarding the schedulability of the system is impossible. We circumvent this issue by assuming worst-case behavior at all times, which at the very least yields no false-positives (cases where schedulability is falsely asserted by the framework).

Additionally, the instruction cycle counts for the ARM Cortex-M7 are, to our knowledge, not public information. This means that for this architecture, the Symex model is based on Cortex-M4 and the results may not be safe to use.

Contrasting this is the RISC-V [29] ISA. The currently modelled RISC-V core is single-stage, with a trivial timing model. We expect that more complex RISC-V implementations run into modelling issues similar to ARM. However, for these, the specification is often more public, often as far as even the HDL implementation being open source [30]. This allows for more precise GA modelling of the architecture, and by extension, tighter WCET bounds.

2) Path explosion: An issue inherent to symbolic execution is that of path explosion. The problem stems from every branch decision yielding (at least) two paths, which leads to exponential growth in the amount of paths to explore. Symex makes no attempt at solving this, arguing instead that in the case of embedded systems, the control flow is simple enough to avoid the issue. It may even be argued that path explosion in an embedded context points to deeper underlying issues in the system design, i.e. runaway cyclomatic complexity, widely cited as an embedded systems design anti-pattern [31], [32].

3) Non-precise floating-point analysis: Many SMT solvers do not support retrieval of the particular encoding of a floating point value once it has been converted in to a floating point operand. This as the IEEE-754 [33] standard does not rigorously define all encodings for all edge-cases. Due to this limitation Symex currently supports three different floating point models, unconstrained, precise but unconstrained once encoding is needed (relaxed) and precise. The precise model is very limited and only suitable for testing where one can be certain that no edge-cases are present, while the unconstrained and relaxed model are generally applicable.

4) Spawn at / after: RTIC provides software tasks as a means to schedule tasks that may not have a fixed period time or event on which they are triggered. EASY supports analysis for software tasks, however, if the task is to be dispatched at some point in the future RTIC uses sorted timer queues [34] to manage the queue of tasks. To properly compute the WCET for a task that enqueues a message in to a sorted timer queue one must explore all possible paths in the queue management, meaning that the problem becomes a complexity analysis of the sorting algorithm, which is not well suited for verification using symbolic execution.

5) Hardware models: The current version of the EASY tool only has partial support for the SAM-{E, S, V}7 family of MCUs. The current implementation of the models does not take into account that the peripheral time delays are not extended when the currently executing task is preempted by some other task. This is not ideal as it will, likely, introduce more overhead than needed. However, such over-approximations make for safe-to-use analysis results for blocking peripheral interactions in hard-real-time systems.

6) Task isolation analysis: Although the memory layout of the resources allotted to a task is technically known at compile time because of their static allocation, the Rust language as it stands is not powerful enough to actually produce these numbers at compile time. Because of this, EASY task isolation analysis incurs some (albeit small, a few instructions per resource) memory overhead, guarded by a branch that is never taken (i.e. disabled) at actual run-time. In case this overhead is unacceptable, the system designer may also decide to omit the analysis specific snippet(s) from production code. This comes with the obvious drawback that the analysis is no longer performed on exactly the production binary.

7) Adherence to requirements: Bounding the behavior definition using assertions as described in IV-E, requires run-time checks to be inserted. Although the analysis guarantees absence of panic, i.e. that these assertions never fail, omitting them from the production binary runs into a problem similar to the task isolation analysis, i.e. the binary under analysis no longer being identical to the production binary.

8) Memory access latency: Symex based WCET estimation currently makes no attempt at modelling memory access latency. This means that the derived WCET results are not valid unless the code is always executing from RAM or if the cache always performs optimally.

B. Future work

In addition to addressing the above limitations, future work includes further verification of the GA ISA models. Current testing builds on hardware-in-the-loop, where code snippets are ran on physical hardware alongside symbolic execution, and comparing the results. This, however can not realistically be made exhaustive, and can only ever suggest the correctness of the GA model. An interesting venue is applying model checking to prove the equivalence of the GA models against a golden reference (e.g. the Spike [35], or Sail [36] RISC-V and ARMv8 models).

VII. Conclusion

This paper has introduced both recent work on the Symex tool alongside the EASY verification framework. We have shown that EASY can correctly verify that tasks only interact with their provided resources thus strengthening the Rust memory safety guarantees in the presence of unsafe code. Further, the paper has shown that Symex can produce safe WCET estimates for Rust code for multiple targets. Building on this, we've shown that EASY can automatically determine the overall schedulability of the system under analysis.

References

[1] E. Serrander, "Worst case execution time estimation using symbolic execution of machine code," 2025.

[2] I. Jönsson, "Easy: Static verification and analysis of rust rtic applications," 2025.

[3] Cristian Cadar, Daniel Dunbar, Dawson Engler & contributors, "KLEE - A Dynamic Symbolic Execution Engine," 2023.

[4] Rust Team, "Rust - A language empowering everyone to build reliable and efficient software," 2024. URL: https://www.rust-lang.org/.

[5] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, pp. 48–62, 2013.

[6] A. Sharma, S. Sharma, S. R. Tanksalkar, S. Torres-Arias, and A. Machiry, "Rust for embedded systems: Current state and open problems," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, (New York, NY, USA), p. 22962310, Association for Computing Machinery, 2024.

[7] T. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, pp. 67–99, Mar. 1991.

[8] Z. Madaoui, H. Lunnikivi, P. Dzialo, and P. Lindgren, "Towards modularity of the rust rtic real-time scheduling framework," in *2024 IEEE Nordic Circuits and Systems Conference (NorCAS)*, pp. 1–7, 2024.

[9] J. Xu and D. Parnas, "Scheduling processes with release times, deadlines, precedence and exclusion relations," *IEEE Transactions on Software Engineering*, vol. 16, no. 3, pp. 360–369, 1990.

[10] J. Eriksson, F. Häggström, S. Aittamaa, A. Kruglyak, and P. Lindgren, "Real-time for the masses, step 1: Programming API and static priority SRP kernel primitives," in *2013 8th IEEE International Symposium on Industrial and Embedded Systems (SIES 2013) : 19-21 June 2013, Porto, Portugal*, pp. 110–113, 2013.

[11] P. Lindgren, P. Dzialo, and H. Lunnikivi, "Hardware support for Static-Priority Stack Resource Policy based scheduling," in *2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE)*, 2023.

[12] A. Nurmi, P. Lindgren, A. Kalache, H. Lunnikivi, and T. D. Hämäläinen, "Atalanta: Open-source risc-v microcontroller for rust-based hard real-time systems," in *Architecture of Computing Systems* (D. Fey, B. Stabernack, S. Lankes, M. Pacher, and T. Pionteck, eds.), (Cham), pp. 316–330, Springer Nature Switzerland, 2024.

[13] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[14] J. Norlen, "Architecture for a Symbolic Execution Environment," 2022.

[15] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, 2004.

[16] C. Österberg, "Calculation of wcet with symbolic execution," Master's thesis, LuleåUniversity of Technology, 2022.

[17] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)* (A. Gupta and D. Kroening, eds.), 2010.

[18] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 53–58, 2014.

[19] A. Niemetz and M. Preiner, "Bitwuzla," in *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II* (C. Enea and A. Lal, eds.), vol. 13965 of *Lecture Notes in Computer Science*, pp. 3–17, Springer, 2023.

[20] C. Ferdinand, "Worst case execution time prediction by static program analysis," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, (Santa Fe, NM, USA), pp. 125–127, IEEE, 2004.

[21] P. Gai, G. Lipari, M. Di Natale, N. Serreli, L. Palopoli, and A. Ferrari, "Adding timing analysis to functional design to predict implementation errors," tech. rep., SAE Technical Paper, 2007.

[22] A. Djoudi and S. Bardin, "BINSEC: binary code analysis with low-level regions," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings* (C. Baier and C. Tinelli, eds.), vol. 9035 of *Lecture Notes in Computer Science*, pp. 212–217, Springer, 2015.

[23] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion, "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pp. 653–656, IEEE Computer Society, 2016.

[24] O. Nicole, M. Lemerre, S. Bardin, and X. Rival, "No crash, no exploit: Automated verification of embedded kernels," in *27th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2021, Nashville, TN, USA, May 18-21, 2021*, pp. 27–39, IEEE, 2021.

[25] G. C. Buttazzo, *Hard real-time computing systems : predictable scheduling algorithms and applications*. Real-time systems series, New York: Springer, 3rd ed. ed., 2011.

[26] Microchip Technology Inc., "32-bit Arm Cortex-M7 MCUs with FPU, Audio and Graphics Interfaces, High-Speed USB, Ethernet, and Advanced Analog."

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

[28] ARM Limited, "Cortex-M4 Technical Reference Manual r0p0."

[29] A. Waterman, K. Asanovic, J. Hauser, and RISC-V International, "The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture," 2019.

[30] R. Hller, D. Haselberger, D. Ballek, P. Rssler, M. Krapfenbauer, and M. Linauer, "Open-source risc-v processor ip cores for fpgas overview and evaluation," in *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–6, 2019.

[31] G. Holzmann, "The power of 10: rules for developing safety-critical code," *Computer*, vol. 39, no. 6, pp. 95–99, 2006.

[32] D. Wallace, A. Watson, and T. Mccabe, "Structured testing: A testing methodology using the cyclomatic complexity metric," 1996-08-01 1996.

[33] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, July 2019.

[34] P. Lindgren, E. Fresk, M. Lindner, A. Lindner, D. Pereira, and L. M. Pinho, "Abstract timers and their implementation onto the ARM Cortex-M family of MCUs," *SIGBED Rev.*, vol. 13, pp. 48–53, Mar. 2016.

[35] "Spike RISC-V ISA Simulator," 2017.

[36] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "Isa semantics for armv8-a, risc-v, and cheri-mips," *Proc. ACM Program. Lang.*, vol. 3, Jan. 2019.