Appears in: Proceedings of the 46th IEEE Real-Time Systems Symposium (RTSS 2025)

Dynamic Fuzzing-Based Whole-System Timing Analysis

Alwin Berger[1], Simon Schuster[2], Peter Wägemann[2], Peter Ulbrich[1]
[1]*Technische Universität Dortmund,* [2]*Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany*

***Abstract*—Worst-case timing analysis traditionally begins with estimating the worst-case execution time (WCET) of individual tasks using either static analysis or measurement-based techniques. To derive worst-case response times (WCRTs), engineers typically compose these WCETs with bounds on preemption and operating system overheads.**

However, WCRTs depend on complex system-level interactions, including task communication, OS behavior, and asynchronous events. Compositional analysis often overestimates, assuming that worst-case conditions across components coincide, admitting infeasible global control-flow paths. Static whole-system techniques refine this by modeling the system holistically but require platform-specific tailoring or extensive annotations. A dynamic equivalent has been missing.

We present FRET, the first dynamic whole-system approach for estimating WCRTs. FRET employs feedback-guided fuzzing to uncover timing-critical dependencies, including inter-task communication, task/OS interactions, and interrupt effects, without requiring prior knowledge of inputs or states. Implemented using LibAFL and evaluated on FreeRTOS with realistic benchmarks, FRET consistently outperforms state-of-the-art fuzzing strategies in estimating accurate response times. Although not sound, FRET delivers more than timing estimates: it produces actionable artifacts—worst-case inputs, interrupt schedules, and inter-task program-flow information—that complement static analyses and support system validation, runtime monitoring, and robust mixed-criticality scheduling.

***Index Terms*—dynamic analysis, measurement-based timing analysis, whole-system analysis, fuzzing, worst-case response time**

I. Introduction

Determining worst-case timing, crucial for ensuring timely operations, poses significant challenges in developing real-time systems. Traditional approaches focus on deriving the worst-case execution time (WCET) (more precisely, an upper bound thereof) of individual tasks using static analysis techniques [1], particularly in hard real-time settings. These techniques are most effective when the system structure is mainly static and the hardware platform relatively deterministic (i.e., expressive hardware models are feasible). Existing timing analysis tools handle task-local path analysis (i.e., value analysis) but do not fully account for OS overheads and asynchronous events such as interrupts [2], [3]. Moreover, static approaches tend to suffer from excessive overestimation when detailed hardware information is unavailable.

In cases where hardware models are unavailable or too pessimistic, the de facto alternative is to apply measurement-based timing analysis (MBTA) techniques [4]–[7]. The industry's reliance on MBTA for early-stage validation [8] and real-time certification, especially in mixed-criticality systems [9], further underscores its practical relevance. Unlike static analysis, MBTA is based on empirical measurements and yields realistic but unsound worst-observed execution times (WOETs) adapted to specific execution scenarios. In practice, however, MBTA still relies on carefully crafted test cases that must trigger worst-case paths through appropriate input selection. Tools such as AbsInt's TimeWeaver [2] and Rapita Systems' RapiTime [3] illustrate how static path analysis can be combined with MBTA to address this input dependence. Still, since measurements substitute the hardware model, the test cases must expose worst-case behavior at the instruction level, burdening their quality and completeness.

Beyond that, engineers are also interested in the worst-case response times (WCRTs) of tasks, which reflect the end-to-end delay from task activation to completion. WCRTs are influenced by local WCETs and system-level phenomena such as OS overheads, resource contention, and asynchronous preemptions. Classical WCRT analysis [10] composes individual WCETs with conservative upper bounds on interference. This compositional approach implicitly assumes the worst-case of each component can coincide, which leads to substantial pessimism. More critically, it admits global control-flow paths that may not be feasible in any concrete execution. One alternative is to fall back on end-to-end measurements to derive a worst-observed response time (WORT). However, this again requires triggering the system with worst-case inputs in worst-case system states, including critical load, interrupt timing and resource-contention patterns.

Static whole-system techniques have been proposed to mitigate pessimism in WCRT analysis. These integrate application-level and OS-level behavior into a unified analysis framework that explicitly models global system control flow. By accounting for scheduling effects, system calls, resource usage, and asynchronous events, such approaches refine local WCETs estimates using system-level flow facts and eliminate infeasible interleavings—resulting in tighter, more realistic WCRT bounds. Despite their analytical power, these tools are challenging to apply in practice. For instance, SysWCET [11] targets custom statically configured OSEK systems and achieves system integration by tailoring source code and system calls. In contrast, SWAN [12] generalizes the approach to dynamic real-time operating system (RTOS) configurations but requires extensive manual annotations across the kernel and application code [13]. Thus, current whole-system analysis approaches are either constrained to niche

platforms or impose significant integration effort, highlighting the need for a dynamic whole-system alternative that retains said benefits while improving generality and automation.

To address these shortcomings, we turned to fuzzing techniques. Its iterative, feedback-driven nature makes fuzzing suited for uncovering rare timing dependencies across system layers. Fuzzing, a technique that has been around since the 1990s [14], is a vital tool within the security community due to its efficacy in exploring complex system behaviors [15]. This success has inspired its application in other domains, including real-time systems. However, dynamic techniques related to fuzzing have been applied only to individual tasks in isolation [6], [16], [17].

In this paper, we present FRET, the first dynamic whole-system analysis approach to Fuzzing worst-observed REsponse Times. FRET provides a gray-box, feedback-driven approach: it does not require a priori knowledge of worst-case inputs, execution paths, system states or even task configurations. Instead, it actively explores the system's control-flow space across all layers, including user tasks, operating system interactions, and asynchronous events, and identifies those conditions that empirically lead to high response times.

A. Problem Statement

In FRET, we must address three key technical challenges:

a) System Tracing and Event Interpretation: Capturing realistic worst-case behavior (i. e., WORT) requires tracing interactions across application tasks, the OS, and asynchronous events. However, to guide fuzzing meaningfully, these traces must be semantically interpreted to expose system-wide dependencies such as data flow, preemptions, and blocking. FRET addresses this by instrumenting all system calls through an OS-specific binding layer. This enables minimal-overhead event-level tracing and system-specific interpretation of execution semantics, including resource usage and scheduling.

b) Inferring Feasible System States: Instruction- or event-level traces quickly become unmanageable due to their sheer volume. Instead, FRET abstracts execution into a compact, semantic model by leveraging the concepts of Atomic Basic Blocks (ABBs) and state-transition graphs (STGs) introduced by the aforementioned static analysis tools [11], [18]–[21]. The ABBs capture coherent control-flow units between system calls, while the STG encodes transitions between system states. This abstraction allows FRET to efficiently record and generalize control-flow structure, task interactions, and timing-relevant behavior.

c) Multi-Objective Fuzzing: Dynamic exploration must optimize not only for high task-local execution times but also for maximal response times. These two objectives can conflict: inputs that maximize WCET for a task may reduce overall interference, leading to local rather than global maxima. FRET tackles this challenge using a bi-objective fuzzing scheme that evolves inputs and interrupt patterns along both dimensions simultaneously. This allows FRET to explore local execution time peaks while also converging on global WCRT scenarios.

B. Contributions & Outline

In summary, we make the following four contributions:

1) *Dynamic WCRT Estimation:* FRET's dynamic whole-system analysis approach systemically infers timing, worst-case input data, and feasible system states.
2) *State-Aware Exploration:* FRET derives a global state-transition model from execution traces, capturing task interactions and OS behavior. FRET explores asynchronous events and their worst-case impact.
3) *Empirical Validation:* We evaluate FRET using various benchmark systems and compare against state-of-the-art fuzzing techniques.
4) *Open-Source Prototype*: We release FRET to support further research on whole-system MBTA analysis.

Importantly, FRET is not a replacement for static WCET analysis or sound WCRT techniques. Instead, it complements existing methods by uncovering hard-to-trigger worst-case interactions and exposing concrete system states and input conditions. This makes FRET particularly useful for empirical validation or use in optimistic runtime monitoring and scheduling scenarios in mixed-criticality systems.

II. Background and FRET's System Model

With FRET, we adopt a whole-system view, capturing application-OS interactions to dynamically infer system-level global control flow to (a) eliminate infeasible execution paths and (b) achieve more accurate WCRT estimates. This section outlines our system model and essential background.

A. OS–State-Aware Timing Analysis

Unlike traditional timing analysis approaches that often abstract away (or ignore) OS interactions, whole-system analysis requires an explicit representation of OS states and their transitions in response to task interactions. A more abstract control-flow model is necessary to manage this complexity, which avoids the excessive detail inherent in typical control-flow graphs (CFGs). ABBs provide a practical abstraction for representing OS-application control flow by grouping basic blocks that span between syscalls (either in user or kernel space). Each ABB is uniquely associated with the application or the OS, maintaining a single entry and exit point. Thus, ABBs effectively *abstract from local control flow* (either user or kernel space) and allow a high-level view of control flow where each ABB executes in an atomic manner (apart from interrupts) on its respective level. Figure 1 (bottom left) illustrates an ABB graph spanning the application logic of the given system with two tasks and an interrupt handler.

To extend the utility of ABBs in whole-system timing analysis, Dietrich et al. [11], [20], [21] introduced the state-transition graph (STG) for OS-level state enumeration, representing feasible system states and transitions between them. Each state captures relevant kernel data, such as ready lists, pending interrupts, and taken resources (illustrated in Figure 1, top right). Transitions in the STG reflect the *global execution paths* initiated by task control flows, syscalls, and asynchronous events, as depicted in Figure 1 (bottom right). This

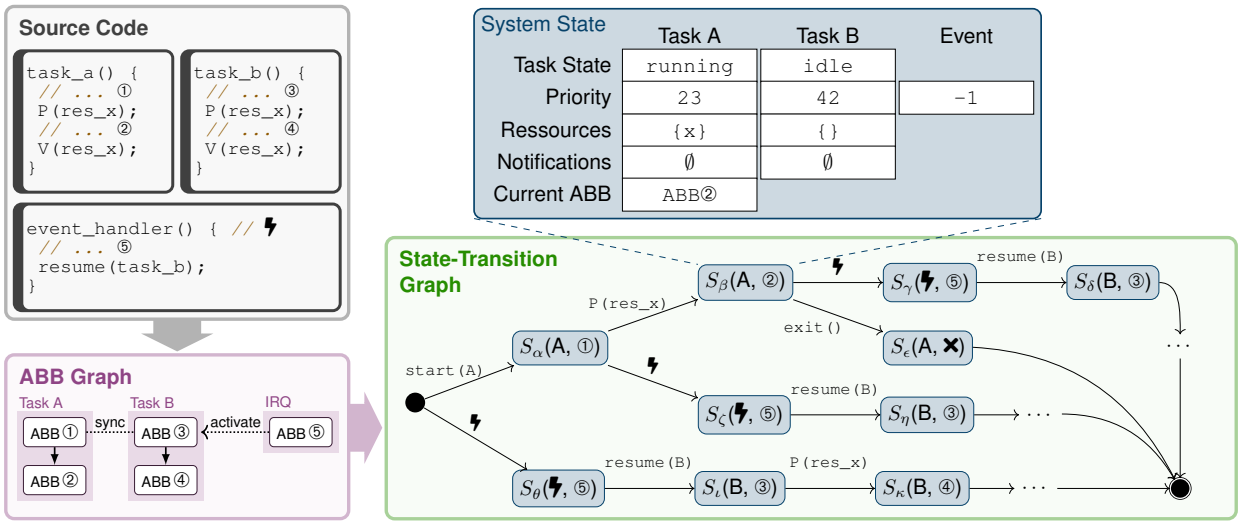


Figure 1: Whole-system analysis abstractions: ABB graph and state-transition graph for two tasks and an interrupt handler. ABBs collapse local control flow that does not alter the system state. STG nodes pair ABBs with their enabling system states, edges induced by system calls or interrupts capture feasible global control flow.

line of work relies on system-state enumeration [20] by static analysis of the application logic (i. e., ABB graph) utilizing the scheduling semantics (i. e., fixed-priority scheduling and resource protocols) and an interrupt model (i. e., minimal inter-arrival times) to statically infer all feasible state S_i from an entry state S_b. A fully enumerated STG contains *all* possible OS states represented as $S = \{S_i \mid S_i \text{ is reachable from } S_b\}$, with transitions $T \subseteq S \times S$. The STG distinguishes transitions by their semantic type: (1) *local (intra-task) transitions* (T_{local}), *dispatch transitions* ($T_{dispatch}$) that dispatch between tasks, and *interrupt transitions* (T_{irq}) that indicate asynchronous events. Each combination of program location (ABB) and system state corresponds to exactly one node. Therefore, the STG contains cycles whenever the same combination of program location and state is encountered multiple times during execution. Since we cannot go into the intricacies further in the paper, we refer to the original work [22] for further details on the construction of the STG.

FRET lifts the STG to a dynamic analysis: at runtime, it discovers system objects, incrementally updates the STG based on observed interactions, and steers feedback-guided fuzzing toward worst-case behavior, enabling the realistic exploration of the feasible state space.

B. System Model

Our approach demands the following properties from the system under test: (1) Data exchange between tasks is made explicit (i. e., system or API calls). (2) Fixed-priority preemptive scheduling and a suitable resource protocol (e. g., PCP [23]). (3) Internal *system state* of the operating system is traceable. (4) Interrupts can be triggered artificially. (5) External inputs (e. g., sensor data) can be simulated.

Some of these, particularly the simulation of inputs, require instrumentation of the target's code to facilitate the fuzzer's interaction with the target.

Most scheduling-relevant system objects, such as tasks, shared resources (e.g., mutexes, semaphores) are dynamically discovered as part of the *system state*, which is extracted at runtime. Thus, only limited information about the system (e. g., allowed interrupt sources and timings and the task to be maximized) needs to be known a priori.

Throughout this paper, we model execution on a single-core system. This aligns with current industry practice in domains like automotive and aerospace, where partitioned scheduling is the prevailing design paradigm and hardware platforms exhibit little cross-core interference and sharing. We discuss the extension to multi-core scheduling in Section V-B.

This work focuses on path aspects and evaluates against manually derived ground truths (WCRTs). Therefore, we rely on an emulated, deterministic processor model that allows for counting instructions. Importantly, FRET is conceptually hardware-agnostic. We revisit accuracy in Section V-C.

III. THE FRET APPROACH

This section introduces FRET, our dynamic whole-system analysis technique. Unlike traditional timing analyses, FRET requires no prior knowledge about worst-case inputs or critical instants of asynchronous events. Instead, it performs system-wide exploration at runtime using feedback-guided fuzzing to uncover execution conditions that lead to WOETs.

We begin in Section III-A by summarizing the input and output of the fuzzing loop. Section III-B details how the system is instrumented and traced to observe control flow, OS interactions, and task states. Section III-C describes how FRET constructs an STG to represent observed global control flow. This graph then informs trace evaluation and feedback in Section III-D. Section III-E covers corpus management to prioritize relevant and diverse inputs. Finally, Section III-F introduces the bi-objective mutation strategy for jointly optimizing local execution costs and global response times.

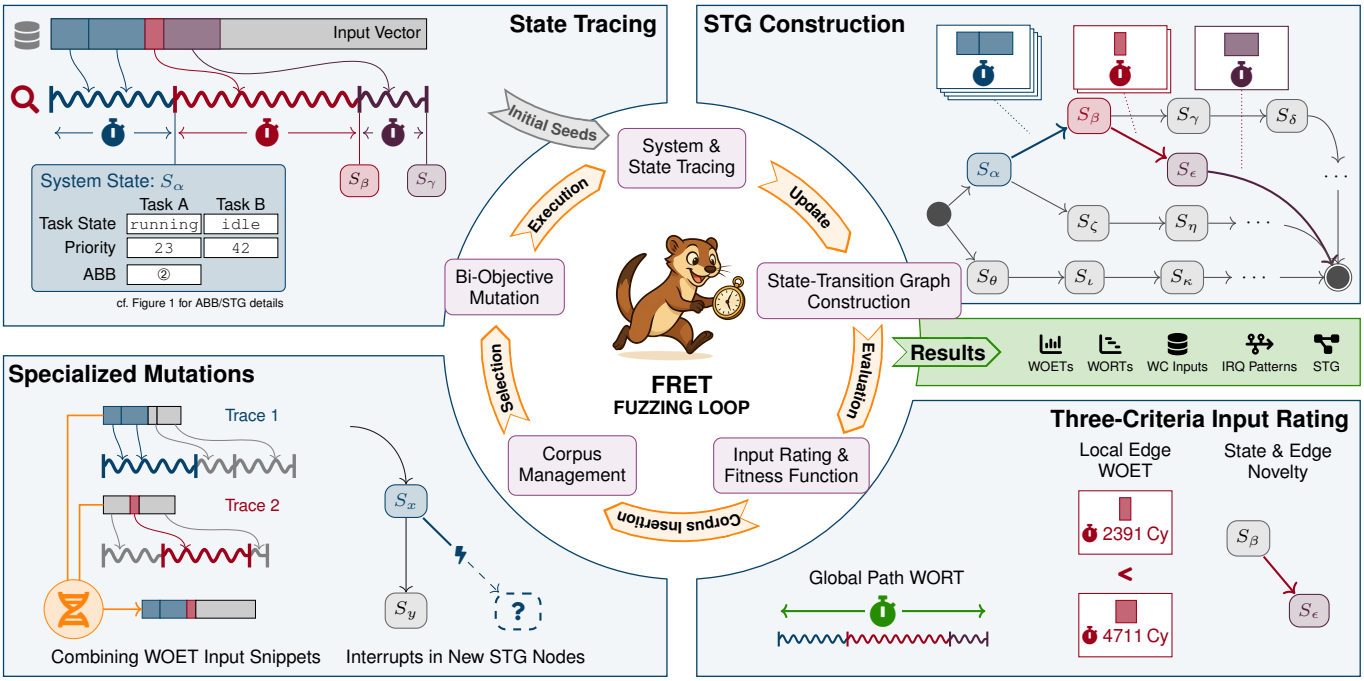


Figure 2: FRET's fuzzing loop. Inputs are loaded, and states are traced during the execution (top left). State traces are inserted into the graph (top right), rated according to different factors (bottom right), and potentially inserted into the corpus. Subsequent inputs are created by mutation, including specialized ones informed by the states (bottom left). As outputs FRET infers WOETs and WORTs, worst-case inputs, interrupt timings, and a STG for downstream analysis.

A. Required Fuzzing Inputs & Analysis Results

FRET is a grey-box approach applicable with minimal system knowledge. The required inputs are: (1) the kernel binary of the target system, (2) a configuration defining the structure and size of the input array, and (3) minimum inter-arrival times for asynchronous interrupts. Only lightweight instrumentation is required: inputs from various sources (e. g., sensors) are simulated by reading from a central data array, and task completions must be signaled to support response time measurement.

The output artifacts of the fuzzing process include: (a) the WORTs of selected tasks, (b) the worst-case input values and corresponding interrupt timings that triggered the worst case, (c) a detailed execution trace of the WORT run, (d) the final state-transition graph representing the feasible global control-flow paths observed during fuzzing, and (e) per-ABB WOET estimates extracted from the STG.

B. System & State Tracing

The heuristics in FRET rely on the system-state transition graph. Thus, tracing system transitions and identifying the active ABB is the first step in FRET's fuzzing loop (see Figure 2). Relevant events include entries and exits of system calls and interrupt handlers and can be traced through software-based instrumentation or hardware-level tracing. We will discuss platform portability and instrumentation strategies in Section V-A. For our prototype, we chose QEMU [24], enabling deterministic timing and full system accessibility without modifying the system under test.

We trace the system under test at discrete snapshots (*snap*), which correspond to state transitions triggered by system calls or interrupts. At each *snap*, FRET records the following data:

- Current time in cycles ($timestamp$)
- Current event ($event$): a tuple of type ($type \in \{syscall\_entry, syscall\_exit, isr\_entry, isr\_exit\}$) and function name ($callname$)
- Source (call site/point of interruption) and target addresses ($addr\_source$, $addr\_target$)
- Current task ($curr\_task$): a task control block from the target system, including the name, priority, state, and pending notifications
- Ready list ($readylist$): all tasks in the $ready$ state
- Delay list ($delaylist$): all tasks in the $blocked$ state
- Consumed input bytes ($inputreads$): a list of tuples of $(byte, addr)$ read since the last snapshot

These snapshots do not contain information about ABBs or individual jobs of each task, which correspond to the execution interval between two snapshots. Each ABB is identified by its start and end point. An application ABB, for example, starts with the return from a system call and ends with another system call, and vice versa for a system call ABB or interrupt handler. Additionally, we must account for preemption, which will cause a preempted ABB to resume later. To address this, we use $addr\_source$ and $addr\_target$ to identify which intervals continue a preempted ABB (abb) and sum up their

execution times (et), as well as their consumed inputs ($bytes$) from all $inputreads$. Apart from ABBs, job instances need to be identified to determine response times, which can be specific to the target system. For our FreeRTOS target [25], we instrumented the application's code to signal a finished job. $readylist$ and $delaylist$ then identify when a new job is released. The result from this stage consists of the snapshots described above, along with information which ABB runs between them and all job releases/responses.

C. System-State Transition Graph Construction

From tracing data, FRET dynamically constructs a system-state transition graph (STG) that captures all observed state transitions during execution, including inter-task data dependencies. Each node in the STG represents a system state combined with the currently executing ABB. Each edge denotes a transition between these nodes and is annotated with relevant timing and input information. Figure 2 (top right) illustrates the trace insertion process and associated timing metadata.

To build the graph from a trace, we define a system state $S_i = (curr\_task_i, readylist_i, delaylist_i, abb_j)$ for each snapshot $snap_i$. The ABB executed between $snap_i$ and $snap_{i+1}$ is denoted as abb_j. Note that abb_j does not necessarily start with $snap_i$ or end with $snap_{i+1}$, because it could have been preempted. abb_j's total execution time (including all segments) is denoted as et_j. $bytes_j$ denotes the set of tuples consisting of all input bytes read by abb_j, along with their memory address. Each state S_i will become a node in the STG. Each edge corresponds to an $event$. Note that all states identical to S_i refer to the same node, which can occur multiple times within a trace.

The graph is constructed iteratively as follows: (1) Let S_k be the last node visited, initially set to the root $S_{initial}$. (2) For each snapshot $snap_i$ with state S_i, check if S_i exists as a node in the graph; if not, insert it. (3) Check for an edge from S_k to S_i exists and is labeled with the triggering event $event_i$; if absent, insert it. (4) If some abb_k was running between $snap_i$ and $snap_{i-1}$, annotate the edge with et_k and $bytes_k$, or update existing values as described in Section III-D. After processing the full trace, we add a terminal edge to a designated S_{end} to mark termination. The resulting STG is the central abstraction driving FRET's state-aware feedback and mutation.

The inferred STG typically under-approximates reachability; nevertheless, empirical convergence can be operationalized via plateaus in STG coverage for a fixed configuration, as demonstrated in Section IV. We discuss convergence and soundness aspects further in Section V.

D. Input Rating & Fitness Function

The feedback function steers fuzzing by rewarding inputs that advance the objective and discarding those that do not. With FRET, we designed our feedback function to focus on multiple instrumental objectives, ultimately aiming to maximize the selected task's response time. The components of the feedback function are illustrated in the lower right corner of Figure 2.

The first instrumental objective is the exploration of new system states, which correspond to differences in the schedule and all inter-task effects. To this end, our feedback function considers a run to be interesting if it caused the insertion of a new node or edge in the graph during the previous step described in Section III-C.

The second instrumental objective is the maximization of local execution times. Increased local execution times increase response times directly and indirectly, that is, by allowing more preemptions. As mentioned, each edge between two nodes S_k and S_l in the graph stores a worst-observed execution time et_k of abb_k. If any of these values have increased, the values of the edge are updated and the input is considered interesting.

Since the overall objective is to increase response times for a selected task, one of our feedback components also rewards this while considering a diverse set of global control-flow paths. Maintaining diversity is important because it is unknown in advance which task interactions lead to the longest response time. It is possible to construct task sets where, for example, a long-running task with high priority is only activated under specific, rare conditions. Our feedback function supports this by recording every path through the STG (i.e., the list of node indices visited) and storing the associated WORT. If an input exceeds the record, it is considered interesting. To eliminate redundant traces, the node indices for each trace are sorted before lookup, making the paths order-independent.

With the STG constructed, we now define how FRET evaluates input quality to guide future exploration.

E. Corpus Management

FRET collects interesting inputs in a corpus, which serves two key roles: selecting the most promising input for the next mutation and pruning obsolete inputs. As the corpus grows, it inevitably accumulates outdated entries that only cover subsets of the graph and offer limited utility for further exploration. To mitigate this, FRET uses a prioritization heuristic.

While response time is the primary criterion for prioritization, it does not ensure path diversity alone. To address this, as noted in Section III-D, each input's (order-independent) path through the STG is used to classify traces. FRET marks the longest-running input for each unique path as a favored input, influencing selection and pruning decisions.

During selection, inputs are weighted by their response times, and one is chosen at random. If this input is favored, it proceeds to mutation. Otherwise, it is likely (e.g., 95 %) to be discarded, and another input is taken. This process maintains focus on high-value inputs without eliminating randomness.

Despite the emphasis on long-running inputs, corpus size can become unwieldy, consuming significant memory and retaining numerous favored inputs linked to obsolete paths. To contain this, FRET periodically prunes the corpus by removing most non-favored inputs if its size exceeds a preset multiple of the number of favored entries (default: 20×). Additionally, a cap on favored inputs (default: top 1 000) eliminates shorter traces, even among the favored set.

F. Bi-Objective Mutation

The mutator is responsible for generating new inputs based on existing ones from the corpus. Fuzzers usually perform mutations using a set of random operations on the input bytes, unless a specialized mutator for the target application (e. g., a grammar-based generator) exists. We designed FRET for the general RTOS domain, meaning that it has an awareness of the system's internal state but remains agnostic to the specific applications running. We exploit this RTOS awareness to implement two distinct mutators that aim to uncover both local and global worst-case timing conditions.

1) Interrupt Mutation: This mutator enhances the state space exploration by strategically modifying interrupt injection timings. Since interrupt handlers can significantly impact system behavior, precise timing is as critical as task activation patterns. Rather than treating interrupt times as generic input values, FRET enforces a per-source n minimum inter-arrival time ($minia_n$) to prevent unrealistic interrupt storms. This is a common assumption since, in real-world applications with strict real-time requirements, minimum arrival times are typically enforced by corresponding guards at the interfaces of the embedded system, for example, using appropriately configurable timer arrays.

Building on this, each interrupt activation time $t_{n,i}$ is confined to a valid interval, typically $[t_{n,i-1} + minia_n, t_{n,i+1} - minia_n]$. The mutator then examines which states occurred in this window and checks if the STG lacks an edge corresponding to the given interrupt. If such a gap is found, the interrupt is rescheduled within that interval to expand the graph potentially.

2) Per Task Input Handling: This mutator aims to maximize the execution time of each ABB by synthesizing the worst-observed local input patterns. It uses the edge annotations in the STG, which record the highest execution time and the corresponding input bytes read by each ABB in its originating node.

When mutating an input, FRET locates the trace's path in the STG, retrieves the annotated bytes for each edge, and injects them into the mutated input at the positions where the corresponding reads were performed initially. This strategy effectively propagates local execution-time increases across different global control-flow paths in the corpus, enhancing the likelihood of encountering global worst-case conditions.

G. Implementation-Related Remarks

We implemented our prototype using LibAFL [26], a modular fuzzer library, in combination with an instrumented version of QEMU [24] for execution. Noteworthy modifications to QEMU and LibAFL are new callbacks for tracing of system calls and interrupts, time measurements using QEMU's instruction counter feature, and interrupt injection according to fuzzing inputs using QEMU timers. Our implementation of the STG relies on abstractions of task control blocks, ready-lists and ABBs, in order to be applicable to a multitude of target systems. Each implementation of the target system needs custom code to extract the relevant information from the target. Currently, the implemented target is FreeRTOS, which is supported using bindings for the data structures (using bindgen[1]), and hand-written functions to read them from the target. The data structures are located using the symbol table of the kernel binary. We discuss FRET's porting potential in Section V-A.

IV. EVALUATION

This section evaluates FRET's ability to maximize WORTs and identify their specific inputs systematically. Thus, we compared FRET against genetic testing and a coverage-guided fuzzing technique. Our experiments focused on three critical aspects to validate the efficacy of FRET: input-data manipulation, interrupt placement, and the interaction of the two.

We outline our evaluation setup and criteria in Section IV-A and describe application scenarios in Section IV-B. FRET is assessed in Section IV-C, with results discussed in Section IV-D.

A. Evaluation Setup

The field of whole-system WCRT fuzzing is relatively new, resulting in a lack of adequate timing analysis tools that employ dynamic testing of entire real-time systems. Accordingly, we implemented multiple candidates for comparison analogous to established dynamic timing analysis techniques. We also utilized LibAFL [26] for these implementations, ensuring that the heuristic component is solely responsible for any disparity in performance between the techniques. A vital commonality is LibAFL's default mutator, which randomly selects from a set of mutations. This set includes various bit operations as well as a crossover mutator that combines random pieces of different inputs. The latter is significant for genetic algorithms. FRET adds mutation stages as described in Section III. The alternative approaches replace FRET's targeted interrupt mutator with its default mutators or randomly select times within the execution interval.

The first candidate is the *evolutionary* approach [27]. It utilizes response time as a fitness function in a genetic algorithm that performs mutations and selects a set of best-performing inputs. We picked a generation size of 100, which performed best in a quick test. The second candidate is the *coverage*-guided approach. This fuzzer uses an adapted edge-based strategy derived from LibAFL's default configuration. The adaptation ensures that every increase in execution counts of edges and response times is rewarded. As a final comparison candidate, we used *randomized* testing without heuristics.

Finally, we manually and thoroughly determined the WCRT of our evaluation scenarios. To this end, we tuned the tasks' execution time simulation to be able to infer the specific worst-case time with respect to the local and inter-task data dependencies. We then manually set the interrupts to trigger the worst-case overlap scenarios. We then manually set the interrupts to trigger the worst-case overlap scenarios. We assume it to be accurate within a few instructions (due to compiler optimizations), but there is no general method

[1] https://github.com/rust-lang/rust-bindgen

to verify this approach. We could have underestimated an interaction within the system. Furthermore, for reference and to showcase pessimism, we composed bounds by traditional response-time analysis (RTA) [28] using presumed WCETs obtained from the previous step. While having confidence in our approach, there is potential for error.

To summarize, we provide the following evaluation scenarios: (1) FRET: our technique, (2) *evolutionary*: evolutionary testing, (3) *coverage*: WCRT-adapted edge-based fuzzing approach, (4) *random*: Monte-Carlo testing using random inputs, (5) *WCRT*: the actual manually determined WCRT, and (6) *RTA Bound*: static bounds composed by traditional response-time analysis [28].

a) Evaluation Criteria: We evaluated the efficacy of the various techniques by measuring the reported WORTs. The evolution of the corpus over time represents a crucial factor that enables fuzzers to identify progressively longer runs. Accordingly, the techniques run for a prolonged duration, with the observed WCRTs recorded for subsequent analysis. We ran each test at least eight times with different (fixed) random seeds to assess relative performance. The subsequent figures report the maximum (colored solid line) and median (colored dashed line) observed WCRT estimate. The maxima show how well the variants perform in a parallel testing campaign, and this value would be used in a development process. The median is an essential measure for assessing the random process in fuzzing. We further provide the presumed *WCRT* as a constant bound (gray dashed line) and the *upper bound* as a number in the graphs.

b) The Fuzzing Platform: We conducted the experiments on a server using an AMD EPYC 9554P with 405 GB RAM. All fuzzing variants were executed in single-threaded mode, with one physical CPU core allocated for each instance. Each scenario was evaluated using the same number of instances per configuration (ten, five in Figure 6). The 24-hour timeout is standard in the general fuzzing domain (see [29]) and was sufficient to reach the presumed WCRT in each case.

B. Application Scenarios

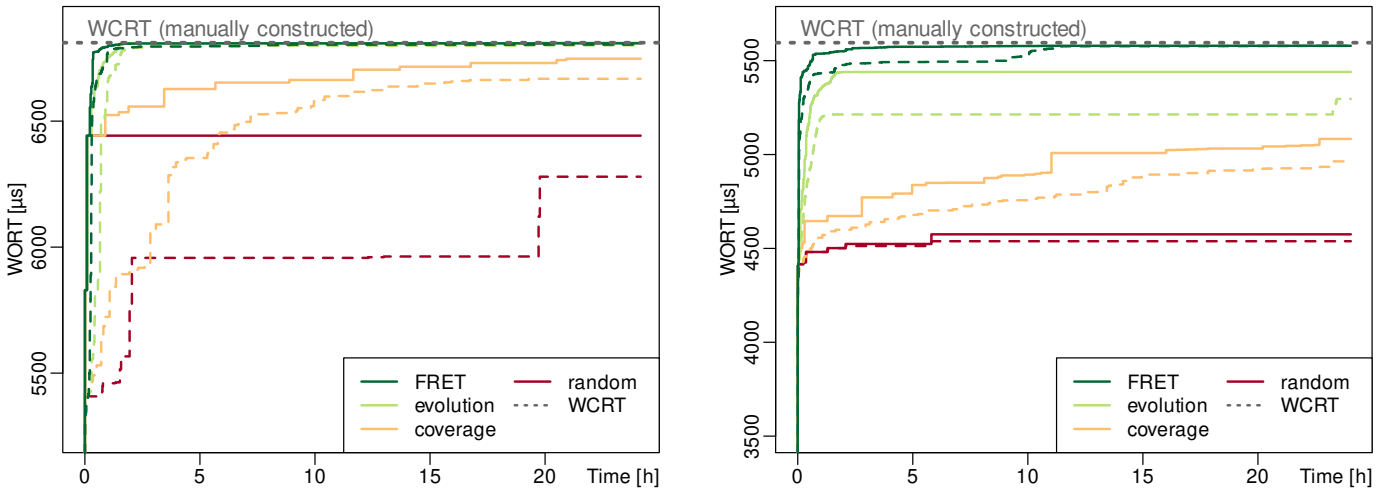
To evaluate our approach, it is necessary to identify appropriate application scenarios for testing that exhibit meaningful interactions between tasks, specifically about data dependencies within tasks, between tasks and the operating system, and between tasks and interrupts. As an execution platform, we selected FreeRTOS [25], running on an (emulated) ARM Cortex-M3 platform (mps2-an385 in QEMU). FreeRTOS is configured to use fixed-priority preemptive scheduling. Multiple tasks of the same priority are handled using round-robin. All input was modeled by reading sequentially from an input array. Communication between tasks is made explicitly using notifications, which are visible in the system state. The computations were simulated using busy-waiting loops, with the iteration count dependent on the input values or values passed from other tasks. In all of the presented scenarios, we selected a single task for which we sought to maximize the WCRT. In the following, we detail our application scenarios:

a) WATERS Industrial Challenge: We used the WATERS Industrial Challenge [30] as the basis for our first benchmark. We reduced the original set by choosing nine periodic tasks (one runnable each), one ISR, and four chains of inter-task communication due to the effort involved in manually translating the model into an actual implementation in FreeRTOS. Each task starts by reading input from the fuzzer, waiting for notifications from chain predecessors, notifying chain successors, and then performing some time-consuming computation depending on these input sources. The objective of the experiments is the third of four tasks in a chain that share a priority level. To compensate for the reduced number of tasks and to induce frequent preemption, we scaled up the execution times until we observed comparable CPU utilization. We simulated data-dependent execution times by setting loop bounds based linearly on the input values and branches on arbitrary conditions. The system also features a sporadic task that directly impacts one of the chains, as well as an indirect effect on the order of execution.

b) UAV: Our second example, named *UAV*, is based on the evaluation in related work on static timing analysis [11]. It simulates the flight control software of a UAV and contains multiple periodic and a single sporadic task. The latter has the lowest priority and, therefore, has a limited impact on the response times of the other tasks. We translated their stubbed code from the OSEK API into an equivalent FreeRTOS system. Again, we chose an arbitrary factor to scale the task's execution times, as their code only contained placeholders with relative computation times. To simulate data-dependent execution times, we let part of each waiting period depend on a polynomial function of the input.

c) Asynchronous Task Release: We constructed the final example with the specific intention of evaluating the effects of interrupt timings. Therefore, we created a set of tasks that share a resource, some of which are activated by an interrupt handler and perform differently depending on the availability of the resource. The WCRT for our chosen task is only reached if it is released by an interrupt handler right after a lower-priority task has taken the resource, causing a priority inversion. To prolong this inversion as much as possible, a second handler needs to activate a different task exactly when the resource is released again.

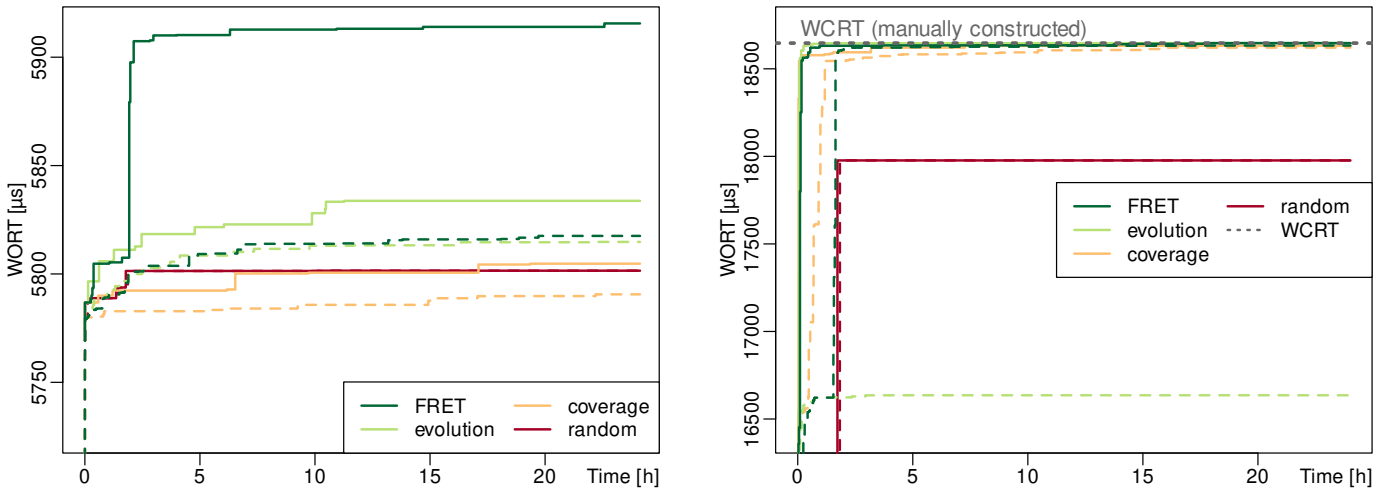
d) Large Automotive: We assessed FRET's scalability using a large benchmark system inspired by automotive control applications. Again, we based the setup on the WATERS task-set generator, which features a real-world industrial configuration shared by an industry partner. The system has 37 tasks in 47 cause-effect chains, with periods of up to 100 ms. Four asynchronous interrupts with defined minimum inter-arrival times (as low as 10 ms) were added to reflect I/O-driven behavior common in automotive systems. These affect some of the chains. To simulate realistic inter-task dependencies, we injected input-dependent control flow. Tasks receive messages whose values influence subsequent paths, and execution times are determined by arbitrary polynomial functions over input data. We set the system load to 90 % utilization.



(a) Scenario: *WATERS*, RTA Bound: 8 212 µs

(b) Scenario: *UAV*, RTA Bound: 11 956 µs

Figure 3: Worst-observed response times with *inter-task data dependencies* but without effects from interrupts. Maximum (solid) and median (dashed) between multiple instances are shown. The known WCRT is marked in gray.



(a) Scenario: *WATERS*, RTA Bound: 8 212 µs, (WCRT construction infeasible)

(b) Scenario: *Asynchronous Task Release*, RTA Bound: 29 483 µs, (WCRT marked in gray)

Figure 4: Worst-observed response times with *interrupts* but without inter-task data dependencies. Maximum (solid) and median (dashed) between multiple instances are shown.

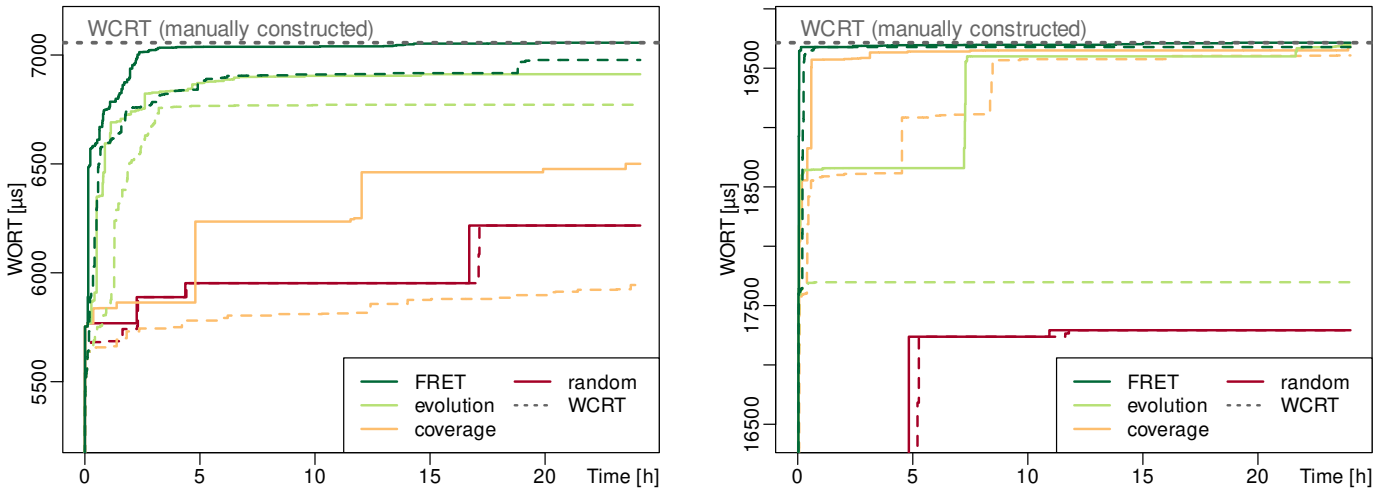
C. Experiments

We conducted experiments to validate our research goals using all analysis variants and application scenarios. These experiments assess FRET's ability to uncover worst-case response times and highlight the role of inter-task dependencies and interrupt timing.

1) Input-Data Assessment: To evaluate FRET's ability to guide input mutations toward long execution paths, we focused on benchmarks with input-dependent execution times and inter-task communication. We used the *WATERS Industrial Challenge* benchmark with interrupts disabled (the sporadic task was made periodic) to isolate data effects. We also evaluated the *UAV* scenario, where interrupt presence does not significantly affect WORT, but data dependencies are more complex. Figure 3 presents the results.

In *WATERS Industrial Challenge*, where dependencies are mostly linear in input values, FRET and evolutionary both converge rapidly. As shown in Figure 3a, FRET reaches saturation early, followed closely by evolutionary, confirming that simple data dependencies are easily exposed.

In contrast, *UAV* features polynomial input relationships, which are harder to explore. Figure 3b shows that FRET is initially the only method to reach the presumed WCRT. Over the whole 24-hour period, both FRET and evolutionary identify this WORT, but FRET exhibits better median performance. We attribute this to FRET's ability to preserve and combine local



(a) Scenario: *WATERS*, RTA Bound: 8 212 µs

(b) Scenario: *Asynchronous Task Release*, RTA Bound: 29 483 µs

Figure 5: Worst-observed response times with *inter-task data dependencies* and *interrupts*. Maximum (solid) and median (dashed) between multiple instances are shown. The WCRT is marked in gray.

worst-case inputs across dependent tasks, which purely local methods often miss.

In conclusion, while multiple techniques could maximize simple data-dependent execution times and inter-task dependencies, FRET was the only one most suitable for more complex data dependencies.

2) Interrupt Timing Assessment: We also need to validate the effectiveness of our state-aware interrupt mutation in isolation. For this purpose, we use the *Asynchronous Task Release* application, with input values fixed at all ones, which is not the worst case in terms of data dependencies. This way, the interrupt timing can only affect the activation of two tasks and, as such, the state in which they are activated. We also use the *WATERS Industrial Challenge* application, with inputs set the same way, leaving only changes in the interrupt timing to influence the order of activation for certain tasks. Figure 4 shows the results. For the *UAV* application in Figure 4b, multiple techniques reached almost the same result, which we confirmed to be the worst possible pattern.

While there is little differentiation between them, *evolutionary* is the only technique that did not find the same pattern in the median case. Even the *random* almost found the worst interrupt timing. Also, FRET was the fastest to find the worst pattern in both the highest and median instance.

Figure 4a shows the same comparison for the *WATERS Industrial Challenge* application. It shows that most instances found the same pattern, but an instance of FRET did find a pattern slightly longer than any other technique. We attribute this to FRET's targeted exploration of all possible system states to interrupt.

Overall, while multiple techniques were able to randomly encounter the worst-case interrupt pattern in one case, FRET found it first and was the only one to reach the true worst case in the other case. These benchmarks suggest that FRET's state-aware interrupt mutation offers an advantage over the default set of mutations that all other techniques utilize.

3) Combined Assessment: For our assessment of simultaneous maximization of input data and interrupt times, we use our *Asynchronous Task Release* and *WATERS Industrial Challenge* applications (*UAV* only has a trivial interrupt and was already shown in Figure 3b). Figure 5b shows that FRET was the only technique to maximize both input components to reach the WCRT on our *Asynchronous Task Release* application. Figure 5a shows the same for our *Asynchronous Task Release* application, with both *coverage* and *evolutionary* close behind in their maxima but further behind in their median.

In previous experiments, we consistently showcased a low-priority task, assuming that its response time had the greatest significance for our evaluation. In order to validate our assessment on a broader set of tasks, we performed an additional evaluation for multiple tasks in our *WATERS Industrial Challenge* example. The results are given in Figure 6. It shows that FRET meets or exceeds the competition for each task.

Overall, this result underlines that FRET's advantages over the other techniques also manifest when both values and interrupt timings are to be mutated. FRET was the only technique in our comparison that consistently found the true WCRT. The *evolutionary* yields times closest to FRET in scenarios where the emphasis lies on input-data dependencies.

4) Scalability Assessment: We used the *Large Automotive* application to test FRET with a large system. This scenario has a much greater number of possible states compared to the previous examples. As a result of this complexity, the STG initially exceeded our system's memory capacity. We identified and excluded two major contributors to the high number of possible states. For one, tasks of the same priority can have a different order in the ready queue, which is affected by the pattern of preemption. We chose to reduce the combinations by ignoring the order of the queues beyond their first element.

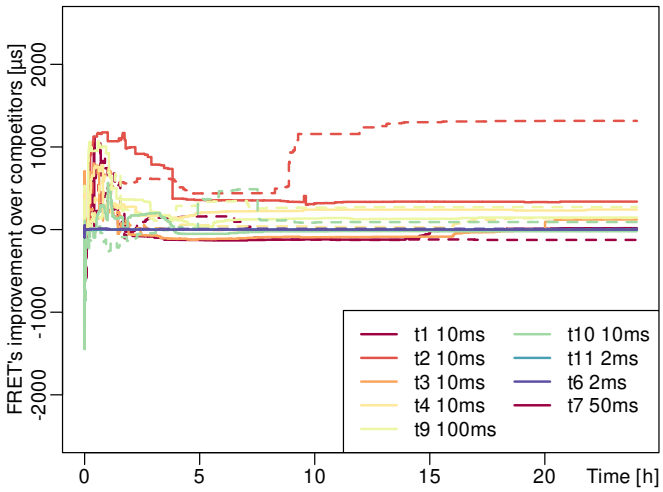


Figure 6: The absolute lead (positive values) or lag (negative values) of FRET for all tasks compared to the best competitor over the course 24 hours. Scenario: *WATERS Industrial Challenge* with *inter-task data dependencies* and *interrupts*.

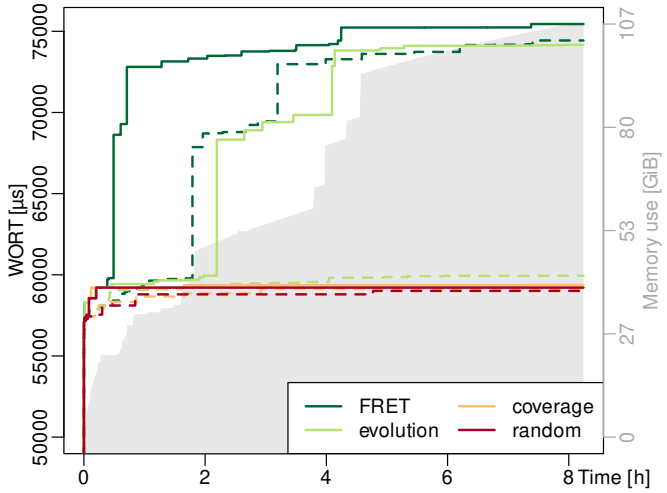


Figure 7: Scenario: *Large Automotive*. Maximum (solid) and median (dashed) between five instances are shown. The gray area represents FRET's memory usage. FRET's analysis is based on a simplified STG.

Another major contributor to the state-space is the value of task notifications, which can be an arbitrary 32-bit value in this scenario. We chose to disregard the value in this case and only respect the notification's status in the state. With these changes, we were able to keep the STG manageable.

The results of a limited eight-hour evaluation with five instances can be seen in Figure 7. Both the runtime and number of instances were reduced due to resource limitations, compared to previous examples. Since there is no tight baseline, this scenario serves to demonstrate the feasibility of FRET's approach and its advantage over other techniques, at least within the initial window of time.

The resulting STG contained 195.7 million nodes and 216.5 million edges, representing 184 962 complete execution paths, including 42 346 order-independent paths. The peak memory use was 106.5 GiB. At this size, the graph remained structurally manageable and usable for feedback-driven fuzzing. Runtime-wise, FRET's heuristics scale is based on the length of the observed global control-flow path, not the number of tasks. Overall, this evaluation scenario demonstrates the computational viability of FRET for larger systems.

D. Summary

Based on our observations from the evaluation scenarios, we draw the following main conclusions: Our evaluation demonstrates that FRET effectively discovers worst-case response times across diverse scenarios. Compared to baseline techniques, it consistently reaches higher WORTs more quickly and reliably, thanks to its integration of system-state modeling, interrupt-aware mutation, and targeted feedback. FRET not only maximizes timing metrics but also reveals the input conditions, execution paths, and system interactions responsible for worst-case behavior. This transparency is essential to understand and validate real-time systems under realistic assumptions. Finally, while FRET imposes substantial memory demands on the fuzzing platform, it scales to system sizes found in practical applications.

V. Discussion & Open Research Questions

FRET demonstrates that dynamic whole-system analysis is feasible. While our evaluation confirms its efficacy and practical utility, several aspects require further exploration:

A. Proving FRET on Other Platforms

FRET inherits MBTA's independence from platform-specific analysis models and the gray-box exploration style of fuzzing: no pre-identification of test cases (i.e., worst-case inputs) is required. FRET contributes a whole-system exploration strategy and an STG-based abstraction that carries these benefits into dynamic timing analysis. Porting FRET to new platforms presents several challenges, including implementing OS-specific bindings, enabling minimally intrusive tracing on hardware, and injecting interrupts with precise timings.

FRET provides an abstraction layer for new RTOSes, with lightweight bindings specifying which functions (e. g., system calls) trigger callbacks and what data is relevant to the analysis. Interpreting this data is OS-specific and occasionally non-trivial (e. g., identifying the originating task during interrupt handling) but remains a one-time engineering task requiring familiarity with the RTOS's data structures and ABI. More broadly, all analysis methods entail setup, dynamic tools need tracing and I/O harnesses; static tools need models and annotations, and FRET's OS/task instrumentation is of comparable effort, after which it runs automatically.

The more significant challenge lies in enabling tracing with minimal probe effect. This is a well-known real-time system issue, and various solutions have been developed. Software-based methods include logging minimal diffs of system state

and timestamps to memory for deferred analysis (e. g., FeatherTrace [31]). On the hardware side, modern embedded platforms often feature built-in tracing support. For example, Intel's Processor Trace is regularly used in fuzzing applications [32] and also embedded platforms such as Infineon's AURIX[2] provide on-chip tracing memory and off-chip transfer capabilities [33] that exceed FRET's requirements. Dedicated hardware tracers, such as Lauterbach's PowerTrace System[3], offer additional options for non-intrusive tracing. These technologies are widely used in industrial settings, demonstrating that effective tracing solutions are readily available.

Implementing precise interrupt patterns is also feasible with existing platforms. Many embedded systems offer complex timer modules that can be programmed to trigger interrupts at specific times. For instance, the Generic Timer Module on Infineon AURIX platforms allows multiple predefined and conditional triggers to be sent to the interrupt controller.

For hardware effects on the STG construction, we assume that cache-related preemption delays will be the primary cause of most variability. Paths through the STG seamlessly encode preemptions, so we expect FRET's feedback function to adapt well, even without any changes. Some adaptations may still be useful. One option would be to include parts of the hardware in the state definition, which, however, would massively increase the state space. Another option would be to store path-dependent execution times in the edges.

In summary, no conceptual obstacles exist to adapting FRET to other RTOSes and hardware platforms. In our evaluation, we chose a QEMU setup not because of any intrinsic limitations of FRET but primarily because the baseline techniques used for comparison are only available in this setting.

B. Towards Multi-Core Analysis

While we implemented FRET for single-core analysis, it already applies to multi-core environments. Many safety-critical real-time systems employ a partitioned architecture, where tasks are statically assigned to cores, allowing each core to be analyzed independently. Partitioning remains the dominant practice, based on our experience with major industry partners.

Existing research efforts, such as ARA [34] and MultiSSE [35], provide a foundation for supporting whole-system analysis of multi-core platforms. MultiSSE, in particular, infers one STG per core and links cross-core nodes between these graphs only at necessary synchronization points, which aligns well with partitioned systems and limits the combinatorial explosion of the state space. We plan to extend FRET similarly by constructing per-core STGs and only tracking synchronization edges where required.

C. Soundness

The aspect of soundness has two dimensions in the context of FRET: First, FRET, like any measurement-based technique, is inherently unsound and does not claim safe upper bounds on WCETs. Instead, it targets practically relevant WORTs by identifying critical input and system-state combinations through guided exploration. The execution time per STG node is derived from empirical measurements, and FRET is agnostic to how these are obtained; we consider this a strength, not a weakness, of FRET. We acknowledge that maximizing task-local WCET and modeling interference patterns (e.g., shared caches, pipelining) is essential for measuring timing accurately, especially in multi-core settings. While this paper focuses on whole-system control and data flow, future versions of FRET may incorporate refined mutators for local execution time maximization based on architectural features. Our current implementation uses a simplified fixed-cost-per-instruction model to maintain a ground truth (actual WCET) for evaluation. This simplification enabled the manual derivation of WCRT by identifying critical input/overlap cases. Introducing a more realistic hardware model would have hindered this reference point, and static analysis would only yield upper bounds with unknown pessimism. Consequently, a direct comparison on real hardware was not possible. Future work could resolve this dilemma using generated benchmarks with known WCET and WCRT properties, which we will discuss in a moment.

Second, FRET's dynamically constructed STG is likely to be unsound, as it under-approximates the whole system state space. It only includes paths observed during execution and thus may omit feasible but unobserved states. This contrasts with static approaches, which are typically over-approximating and include infeasible paths. We argue, however, that STG soundness is a solvable problem. FRET can actively explore and invalidate infeasible transitions from a statically inferred STG (e.g., using ARA). Achieving this requires extending FRET with classification techniques and probing strategies to target statically identified but dynamically unconfirmed paths. Should the two representations match, we would have obtained a sound and complete STG for the system under analysis. To reiterate, FRET is not a substitute for sound analysis but complements it. It helps developers explore system behavior efficiently and identify meaningful timing scenarios. This supports downstream use cases such as further verification, mixed-criticality scheduling, and runtime monitoring based on observed worst-case states and inputs.

D. Synthesis of Benchmark Systems with Known WCRTs

We consider the aforementioned dilemma of determining FRET's accuracy a fundamental challenge [36], [37]: Comparable to WCET analysis of tasks in isolation, determining this ground truth WCRT is inherently not possible based on existing benchmarks [36]. An avenue for future work is constructing benchmark systems with known worst-case inputs, task interactions, OS overheads, and interrupt patterns to serve as reliable ground truth. However, the design and implementation of such a whole-system generator is a non-trivial research and engineering challenge that requires further investigation. In particular, one critical step is synthesizing functional task code that exhibits specific execution times while preserving the intended worst-case control-flow path

[2]Infineon AURIX TC3xx Family: https://www.infineon.com
[3]Lauterbach PowerTrace II: https://www.lauterbach.com

under realistic hardware conditions. This includes ensuring that, even in the presence of cache effects or cross-core interference, the synthetically defined worst-case execution path remains dominant.

VI. Related Work

A huge body of related work exists for dynamic timing analyses and response times. However, to our knowledge, FRET advances the state-of-the-art with regard to its (1) use of fuzzing techniques, (2) notion of systems states capturing the interaction between tasks/OS, and (3) handling asynchronous events. Subsequently, we outline relevant areas of related work: genetic algorithms for WCET analysis (Section VI-A), the timing analysis of combined application and operating-system code (Section VI-B), and the trend towards hybrid timing analysis (Section VI-C).

A. Dynamic WCET Analysis & Profiling

The use of genetic algorithms for WCET analysis was introduced in the late 1990s by Wegener et al. [17], [27], being a specific conceptual strategy of the overarching topic of fuzzing. Outside of the real-time domain, software testing approaches such as GA-Prof [38] utilize genetic algorithms for input-sensitive application profiling. They also aim to identify input values that maximize execution times and thus uncover performance bottlenecks. The prominence of machine learning techniques besides genetic algorithms also extends into the field of timing analysis. For example, a recent report by the FAA presents a machine learning approach, which first trains a model to predict execution times and then uses it to generate inputs that are predicted to maximize the execution times [39]. In contrast to these works for the domain of WCET analysis of single tasks, FRET addresses the more complex problem of WCRT analysis in real-time applications with multiple tasks. Additionally, the handling of interrupts, inherent to cyber-physical systems, is a further unique characteristic of FRET in the domain of worst-case timing analysis.

B. Integrated Analysis of Applications & Operating Systems

Schneider's static-analysis methodology [40], [41] kicked off OS-aware WCRT analysis. SysWCET [11], proposed by Dietrich et al., was the first to systematically leverage OS-state transition graphs for fixed-priority, statically configured real-time systems. Schuster et al. [12] generalized this idea for generic RTOSes. This work employs annotations [13] in the application and operating system kernel to infer the calling context of system calls. Other researchers use WCET techniques for analyzing operating systems [42]–[44], where we refer to the survey from Lv et al. [45] for details. In contrast to these works, FRET employs a fuzzing-based approach for hardware platforms for which the development of accurate models is unrealistic.

On the side of hardware-agnostic program-path analysis, companies start to integrate the operating-system semantics when finding bugs in safety-critical applications [46], [47]. Sharing this goal, FRET uses a notion of system states along with possible interrupt occurrences in order to analyze the WCRT of applications running on top of an RTOS. High-level context information on global control flow (i. e., flow facts) can be inferred [11] from and annotated [12] to STGs, supporting analyses downstream in the real-time systems' development process.

C. Hybrid Timing Analysis

With the problem of WCET tools being limited by their hardware model, *hybrid* timing analysis is one possible direction to yield timing estimates for non-statically-predictable platforms. The *TimeWeaver* tool [2] is one commercially available example of such types of analyses. The tool uses information from measurement traces and the gathered information on timing costs for basic blocks for static program-path analysis. Having such a combination of measurement-based and static analysis is one direction of future work on FRET. In contrast to *TimeWeaver*, FRET can handle asynchronous interrupts as part of its notion of system states. Thus, this advantage enables us to find more accurate timing estimates in a shorter period of analysis time, which in turn may benefit the accuracy of hybrid timing analysis.

VII. Conclusion

We introduced FRET, the first dynamic fuzzing-based whole-system analysis framework for estimating worst-case response times. Unlike traditional static or compositional WCRT analysis, FRET explores system-wide behavior, including task interactions, OS effects, and asynchronous events, without requiring prior knowledge or manual configuration.

FRET uses feedback-guided fuzzing and system-state modeling to uncover critical timing scenarios that static methods may over-approximate. Its dynamic state-transition graph captures feasible global control flows and guides input mutation and interrupt scheduling to maximize response times. More than just timing estimates, FRET provides execution traces, triggering worst-case inputs, interrupt patterns, and the STG, a semantic model of observed system behavior. These artifacts support empirical validation, runtime monitoring, mixed-criticality scheduling, and can support analyses downstream in the development process. Evaluation showed that FRET outperformed state-of-the-art fuzzers in WORT discovery and convergence. Released as an open-source prototype atop LibAFL and FreeRTOS, FRET invites further research on dynamic whole-system timing analysis.

In short, FRET complements sound analysis by offering practical insights, making it a valuable tool for the design of real-time systems.

The source code and the artifact evaluation of FRET *are publicly available:*
https://git.cs.tu-dortmund.de/SYS-OSS/FRET

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[2] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand, "TimeWeaver: A tool for hybrid worst-case execution time analysis," in *Proceedings of the 19th International Workshop on Worst-Case Execution Time Analysis (WCET '19)*, 2019, pp. 1:1–1:11.

[3] S. Law, M. Bennett, S. Hutchesson, I. Ellis, G. Bernat, A. Colin, and A. Coombes, "Effective worst-case execution time analysis of DO178C Level A software." *Ada User Journal*, vol. 36, no. 3, 2015.

[4] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," *Leveraging Applications of Formal Methods, Verification and Validation*, vol. 17, pp. 430–444, 2008.

[5] F. J. Cazorla, J. Abella, J. Andersson, T. Vardanega, F. Vatrinet, I. Bate, I. Broster, M. Azkarate-Askasua, F. Wartel, L. Cucu *et al.*, "PROXIMA: Improving measurement-based timing analysis through randomisation and probabilistic analysis," in *Proceedings of the Euromicro Conference on Digital System Design (DSD '16)*, 2016, pp. 276–285.

[6] R. I. Davis and L. Cucu-Grosjean, "A survey of probabilistic timing analysis techniques for real-time systems," *LITES: Leibniz Transactions on Embedded Systems*, pp. 1–60, 2019.

[7] F. J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega, "Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–35, 2019.

[8] D. Kästner, M. Pister, and C. Ferdinand, "Obtaining DO-178C Certification Credits by Static Program Analysis," in *Proceedings of the 11th Embedded Real-Time Software Congress (ERTS '22)*, Jun. 2022.

[9] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 82:1–82:37, 2017.

[10] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[11] C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann, "SysWCET: Whole-system response-time analysis for fixed-priority real-time systems," in *Proceedings of the 23rd Real-Time and Embedded Technology and Applications Symposium (RTAS '17)*, 2017, pp. 37–48.

[12] S. Schuster, P. Wägemann, P. Ulbrich, and W. Schröder-Preikschat, "Proving real-time capability of generic operating systems by system-aware timing analysis," in *Proceedings of the 25th Real-Time and Embedded Technology and Applications Symposium (RTAS '19)*, 2019, pp. 318–330.

[13] S. Schuster, P. Wägemann, P. Ulbrich, and W. Schröder-Preikschat, "Annotate once – analyze anywhere: Context-aware wcet analysis by user-defined abstractions," in *Proceedings of the 22nd International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'21)*, 2021, pp. 54—-66.

[14] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," vol. 33, no. 12, pp. 32–44.

[15] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[16] F. Mueller and J. Wegener, "A comparison of static analysis and evolutionary testing for the verification of timing constraints," *Real-Time Systems*, vol. 21, no. 3, pp. 241–268, 2001.

[17] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres, "Testing real-time systems using genetic algorithms," *Software Quality Journal*, vol. 6, no. 2, pp. 127–135, 1997.

[18] F. Franzmann, T. Klaus, P. Ulbrich, P. Deinhardt, B. Steffes, F. Scheler, and W. Schröder-Preikschat, "From intent to effect: Tool-based generation of time-triggered real-time systems on multi-core processors," in *Proceedings of the 19th IEEE International Symposium on Real-Time Distributed Computing (ISORC '16)*, 2016, pp. 134–141.

[19] F. Scheler and W. Schröder-Preikschat, "The real-time systems compiler: migrating event-triggered systems to time-triggered systems," vol. 41, no. 12, pp. 1491–1515, 2011.

[20] C. Dietrich, M. Hoffmann, and D. Lohmann, "Cross-kernel control-flow-graph analysis for event-driven real-time systems," in *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*, 2015, pp. 6:1–6:10.

[21] ——, "Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, pp. 35:1–35:25, 2017.

[22] C. Dietrich, M. Hoffmann, and L. D., "Global optimization of fixed-priority real-time systems by rtos-aware control-flow analysis," no. 16, pp. 35:1–35:25, 2017.

[23] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[24] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX annual technical conference, FREENIX track*, vol. 41, 2005.

[25] "Freertos – real-time operating system for microcontrollers and small microprocessors." [Online]. Available: https://freertos.org/

[26] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A framework to build modular and reusable fuzzers," in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS '22)*, 2022.

[27] F. Mueller and J. Wegener, "A comparison of static analysis and evolutionary testing for the verification of timing constraints," *Real-Time Systems*, vol. 21, no. 3, pp. 241–268, 2001.

[28] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61.

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[30] A. Hamann, D. Dasari, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein, "Waters industrial challenge 2017," in *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS '17)*, 2017. [Online]. Available: https://www.ecrts.org/forum/download/WATERS2017_Industrial_Challenge_Bosch.pdf

[31] B. Brandenburg and J. Anderson, "Feather-trace: A lightweight event tracing toolkit," in *Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '07)*, 2007, pp. 19–28.

[32] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 167–182.

[33] M. Brand, A. Mayer, and F. Slomka, "NITRO: Non-intrusive task detection and monitoring in hard real-time systems," in *Proceedings of the 28th International Conference on Real-Time Networks and Systems (RTNS '20)*, 2020, pp. 78–88.

[34] G. Entrup, A. Kässens, B. Fiedler, and D. Lohmann, "Applied static analysis and specialization of cross-core syscalls for multi-core autosar os," *Real-Time Systems*, vol. 60, no. 3, pp. 491–533, 2024.

[35] G. Entrup, B. Fiedler, and D. Lohmann, "MultiSSE: Static syscall elision and specialization for event-triggered multi-core rtos," in *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '23)*, 2023, pp. 262–275.

[36] P. Wägemann, T. Distler, C. Eichler, and W. Schröder-Preikschat, "Benchmark generation for timing analysis," in *Proceedings of the 23rd Real-Time and Embedded Technology and Applications Symposium (RTAS '17)*, 2017, pp. 319–330.

[37] B. Lesage, D. Griffin, F. Soboczenski, I. Bate, and R. I. Davis, "A framework for the evaluation of measurement-based timing analyses," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS '15)*, 2015, pp. 35–44.

[38] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating performance bottleneck detection using search-based application profiling," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 270–281.

[39] B. Andersson, D. de Niz, G. Moreno, J. Hansen, M. Klein *et al.*, "Assessing the use of machine learning to find the worst-case execution time of avionics software," Tech. Rep., 2023.

[40] J. Schneider, "Why you can't analyze RTOSs without considering applications and vice versa," in *Proceedings of the 2nd Workshop on Worst-Case Execution Time Analysis (WCET '02)*, 2002, pp. 79–84.

[41] ——, *Combined schedulability and WCET analysis for real-time operating systems*. Shaker, 2003.

[42] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *Proceedings of the 32th Real-Time Systems Symposium (RTSS '11)*, 2011, pp. 339–348.

[43] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper, "Static Timing Analysis of Real-Time Operating System Code," in *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '04)*, 2004, pp. 146–160.

[44] A. Colin and I. Puaut, "Worst-case execution time analysis of the RTEMS real-time operating system," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS '01)*, 2001.

[45] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang, "A survey of WCET analysis of real-time operating systems," in *Proceedings of the International Conference on Embedded Software and Systems (ICESS '09)*, 2009, pp. 65–72.

[46] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand, "Taking static analysis to the next level: Proving the absence of run-time errors and data races with Astrée," in *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS '16)*, 2016, pp. 570–579.

[47] D. Kästner, A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, A. Schmidt, H. Hille, S. Wilhelm, and C. Ferdinand, "Finding all potential runtime errors and data races in automotive software," in *SAE World Congress*, 2017.