HESTIA: Automated Application Code Identification in RTOS Firmware

Constantin Brinza[1], Peng Liu[2], Jorik van Nielen[1], Daan Prinsze[3], Marius Muench[2], and Andrea Continella[1]

[1] University of Twente [2] University of Birmingham [3] TU Delft
{c.brinza,j.j.vannielen,a.continella}@utwente.nl,
pxl356@student.bham.ac.uk, m.muench@bham.ac.uk,
daanprinsze@gmail.com

Abstract. Embedded devices are increasingly affordable and widely deployed in safety-critical and commercial domains. Yet, they remain vulnerable to firmware-level attacks. ARM Cortex-M microcontrollers dominate resource-constrained systems and are commonly deployed with firmware leveraging real-time operating systems (RTOSs) distributed in a monolithic binary image. Most security-critical vulnerabilities reside in application code, making the identification of such application entry points a prerequisite for meaningful analysis. This process is, however, particularly challenging for RTOS firmware images, which lack debug symbols and clear separation between system and application code. To address this challenge, we perform a systematic study of RTOS frameworks and analyze their initialization mechanisms. Based on our study, we design and implement HESTIA, a novel analysis tool that enables fully automated identification of application entry points in RTOS firmware without requiring any system knowledge. We evaluate HESTIA on three datasets totaling 145 firmware images, demonstrating its effectiveness across a diverse set of firmware images deploying popular RTOSs.

Keywords: Monolithic Firmware · Firmware Analysis · Reversing.

1 Introduction

Connected embedded devices—commonly referred to as Internet of Things (IoT) devices—are ubiquitous and underpin a wide range of systems, including smart homes, industrial automation, and medical devices. The number of IoT devices is projected to reach 40.6 billion by 2034, compared to an estimated 19.8 billion devices in 2025 [27]. Nonetheless, the firmware running on IoT devices often presents security vulnerabilities [2],[7], which are regularly exploited for distributed denial-of-service (DDoS) campaigns [3],[25], unauthorized network intrusions [43], and safety-critical failures [31]. Besides, empirical studies indicate that device manufacturers prioritize rapid product development over long-term maintenance, leaving deployed devices vulnerable to unpatched security flaws [14]. This problem is exacerbated by the long lifetimes of embedded systems and the difficulty of retrofitting security mechanisms post-deployment.

While prior work has focused on automated security analysis of Linux-based firmware—leveraging process boundaries, OS abstractions, and debug symbols—comparatively little attention has been paid to non-Linux firmware, particularly monolithic firmware images interleaving code and data in a single binary blob. In this context, IoT firmware frequently builds upon specialized real-time operating systems (RTOSs) to match design constraints and real-time requirements while offering ease of use for firmware developers. The wide adoption of RTOS-based firmware poses unique security challenges, as RTOSs typically lack process isolation, rely on shared memory, omit basic defenses against memory corruption attacks, and tightly interleave "system services" with application logic [6,36,44,1].

Existing analysis tools for RTOS firmware lack mechanisms to effectively isolate application-level logic from the surrounding system-level infrastructure. As a result, this limitation significantly hinders the effectiveness and scalability of security analyses such as symbolic execution, fuzzing, and static analysis, as these techniques are forced to deal with large volumes of well-tested RTOS kernel code and hardware platform abstraction layers alongside security-relevant application logic. In addition to the lack of symbols and code boundaries, firmware diversity further complicates analysis: different frameworks employ distinct initialization mechanisms, memory allocation strategies, and application entry point conventions. As a result, identifying where application code begins—and what code implements security-relevant behavior—remains a major obstacle.

To address this problem, we propose Hestia, a novel approach for automatically identifying application code in monolithic RTOS firmware images. Hestia identifies application logic by recovering user-defined tasks and their entry functions. Our approach operates without prior knowledge of the underlying OS and requires no symbols or structured binary format. The ability to automatically isolate application code within RTOS firmware ultimately enables targeted, scalable security analysis, allowing vulnerability discovery techniques to focus computational resources on proprietary application logic, and improving the effectiveness of manual reverse engineering, security audits, post-deployment vulnerability assessment, and patch development for legacy devices.

Our approach leverages the observation that, across diverse RTOS implementations, application code is consistently introduced into the system through *task-creation* mechanisms. By identifying task-creation functions and analyzing their call sites, Hestia recovers task entry functions and associated task descriptors. Despite RTOS heterogeneity, these descriptors exhibit common structural properties—e.g., references to a task entry function and optional task metadata—that enable reliable identification of application-level logic.

We evaluate Hestia on 95 RTOS-based and 50 bare-metal firmware images. Our evaluation shows that Hestia reduces the number of functions requiring manual inspection by more than 99%, enabling scalable and targeted security analysis of RTOS-based firmware.

In summary, we make the following contributions:

– We perform a systematic study of user-task initialization patterns and structures in open-source RTOSs.

- We propose a novel, pattern-based approach to identify application code in RTOS-based firmware.
- We implement Hestia, an automated analysis framework that identifies user tasks and their entry points in RTOS-based firmware images.
- We evaluate Hestia on three datasets combining 145 firmware images, demonstrating the effectiveness of our tool.

To foster reproducibility, we open-source our implementation and evaluation datasets at: https://github.com/utwente-scs/hestia.

2 Background & Motivation

Embedded devices run on *firmware* — software that combines high-level application and low-level hardware code. Previous work classifies firmware in three categories based on the used OS type [34,16]. Type-I systems retrofit general-purpose OSs, such as Linux, for the embedded context, Type-II systems leverage real-time operating systems, and Type-III systems run *bare-metal* firmware executing directly on the hardware without any operating system abstractions.

We focus on Type-II systems, which run RTOS system-level code and user-defined applications as a monolithic program in a flat, physical address space.

2.1 RTOS-based Firmware

RTOSs are lightweight operating systems designed for embedded environments. They rely on deterministic scheduling (e.g., preemptive priority or rate-monotonic scheduling) to provide bounded execution guarantees [29]. Beyond task scheduling, RTOSs commonly include functionality for integrated communication, I/O, memory management, and interrupt handling. Just as for bare-metal firmware, all functionality is fit into a single monolithic binary, resulting in little to no separation between user and system components.

Hardware-Abstractions Layers. RTOSs are commonly designed to run on a wide range of different hardware platforms. To facilitate unified programming interfaces, RTOSs are often combined with platform-specific Hardware Abstractions Layers (HALs). HALs interact directly with the underlying hardware, and provide accessible interfaces for both RTOS- and application-layer code, abstracting the intrinsics of the given hardware platform. For most RTOSs, the HAL, RTOS code, and developer-defined application code are compiled into a single monolithic binary.

RTOS Tasks. RTOS-based firmware typically splits the application code in multiple concurrent tasks. For most RTOSs, tasks correspond to kernel-managed threads of execution and represent the primary unit of scheduling [8]. At runtime, a task is represented by a task object comprising the following components:

- **Task Descriptor**: a task control data structure residing in RAM and managed by the RTOS scheduler. It holds task metadata and runtime information, such as state (i.e., whether a task is ready, running, or suspended), and current priority.

- **Task Stack**: a memory region in RAM used for task-local variables and function calls, accessed via the stack pointer.
- **Task Entry Function**: program code residing in ROM that defines the task's execution logic.

Memory Layout. Task descriptors, data structures, and function pointers used to register user tasks are often placed in either RAM-mapped or flash-mapped memory sections, such as `.data` or `.rodata`. The task stack is provided as a linker-visible object residing in RAM, most commonly placed in the `.bss` section or in a dedicated stack section defined by the linker script. At runtime, the RTOS completes initialization of the descriptor using scalar values supplied at task-creation time, compile-time configuration parameters, and pointers to constant objects located in read-only memory (typically `.rodata`), such as the task entry function and task name. For dynamically allocated tasks, the task stack is allocated at runtime from an RTOS-managed heap or memory pool.

Task Creation. Developers define and initialize tasks via the RTOS Application Programming Interface (API). Task-creation API calls allow to specify the entry function address, name, priority, and other RTOS-specific attributes.

To initialize the task, the RTOS will assign memory for the task descriptor and task stack. Task descriptors and task stacks are allocated either statically or dynamically, depending on the RTOS implementation and configuration. When statically allocated, task descriptors and stacks are assigned in a pre-defined region in RAM at link time, while when dynamically allocated, the RTOS-managed heap or memory pool is leveraged for allocation during runtime.

After allocation, the RTOS finalizes task initialization by populating the Task Descriptor with values supplied to the task-creation API (e.g., entry function and task name) and compile-time RTOS configuration parameters. It then registers the task with the scheduler by inserting it into a scheduler-managed data structure (e.g., a ready queue or priority list), an operation performed atomically to prevent concurrent access.

2.2 Motivation

Resource-constrained embedded devices often rely on RTOS-based firmware, which is typically distributed as a stripped, monolithic binary. In this setting, compile-time information, such as function names, memory segments, location of objects in RAM, or even the load-address of the binary is lost. Additionally, application code is tightly interwoven with RTOS frameworks, hardware abstraction layers, and supporting libraries, with no explicit separation between operating system functionality and device-specific application logic.

Although firmware images may contain hundreds or thousands of functions, only a small fraction implements application-specific behavior. This application code constitutes the primary attack surface and is the most likely location for undiscovered vulnerabilities, as it is often proprietary, less scrutinized, and tailored to device-specific requirements. In contrast, RTOS kernel components

and hardware abstraction layers are typically reused across devices and benefit from extensive testing and community review.

Existing automated firmware analysis techniques struggle to operate effectively in this setting. Many approaches either treat all code uniformly or rely on input-driven heuristics to prioritize analysis [9], leading to inefficient use of computational resources, inflated false-positive rates, and missed vulnerabilities in application logic that is not directly exposed to external inputs. These limitations are particularly pronounced in RTOS-based firmware, where the absence of process boundaries and symbols makes it difficult to distinguish application code from framework-supporting components.

The lack of automated mechanisms to identify application code also significantly hampers manual reverse engineering. Analysts are forced to expend substantial effort disentangling security-relevant logic from the broader infrastructure code provided by RTOSs and HALs before meaningful vulnerability analysis can begin. As a result, both automated and manual vulnerability discovery, triage, and patching are fundamentally constrained by the inability to isolate application-level code within firmware [23].

This observation motivates the need for automated techniques that reliably identify application code in RTOS firmware binaries and enable security analyses to focus on the portions of the firmware that are most relevant.

3 Application Code Initialization

We investigate common patterns that enable the identification of application code for RTOS firmware, i.e., the *user tasks* and their *entry functions*. We analyze and reverse seven RTOSs to study the task initialization mechanisms and identify common patterns. For this, we choose popular open-source RTOSs widely used in the industry: FreeRTOS [19], Contiki-NG [11], LittleKernel [28], Zephyr [38], NuttX [18], RIOT [37], and Mbed OS [4]. We specifically identify user tasks. Some RTOSs use tasks to launch system services (e.g., daemons), but since these tasks do not contain application code they are excluded.

We identify three task initialization models:

- **Direct User Task Initialization Model**. The firmware features a *main function* carrying out task initialization before entering a *super-loop* implementing task scheduling and firmware state maintenance.
- **Delegated Initialization Model**. The RTOS initializes a *main task*, which initializes all user tasks, before finishing execution. The main application code is embedded within the different user tasks.
- **Bootstrap-Centric User Task Initialization Model**. Again, the RTOS initializes a main task which initializes all user tasks. Afterwards, the main task will call its `main` function which also contains application code and is periodically scheduled together with user tasks.

Figure 1 illustrates the three RTOS task initialization models; concrete RTOS initialization call graphs are provided in Appendix B. Execution always starts

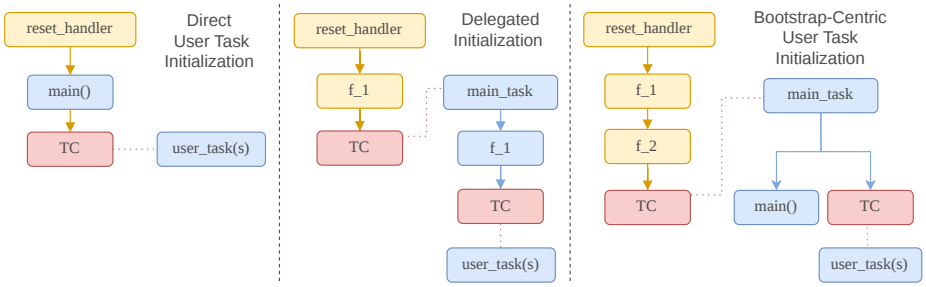

Fig. 1: Schematic overview of three identified RTOS task initialization models. Here, `f_n` denotes call-graph levels from the entry point or main task and `TC` marks task-creation APIs. Solid arrows (→) indicate direct function calls, while dotted arrows (⋯) indicate task creation. Yellow highlights the function call chain leading to the first initialized task, red denotes `TC` functions, and blue denotes application code.

with the `reset_handler` function. Functions deeper in the call graph are denoted as `f_n`, where `n` represents the call-graph level, starting from the `reset_handler` or the main task. A task-creation function is labeled `TC`. Task descriptor data is passed to the `TC` function, either as individual pointers or a single data structure pointer, to initialize the task.

Based on this classification, we group RTOSs according to their implementation. We observe that Mbed OS, RIOT, Zephyr, and NuttX [1] follow the Bootstrap-Centric User Task Model; LittleKernel the Delegated Model; while Contiki and FreeRTOS adhere to the Direct User Task Model. Finally, we note that the `TC` function responsible for initializing the main task may differ from the `TC` function used to initialize other user tasks. While this analysis has been performed on a subset of all available OSs, the notion that these systems require `TC` functions to initialize tasks is extendable to other RTOSs.

4 Design

HESTIA leverages static-symbolic analysis to identify user tasks and their entry points in RTOS-based firmware. Figure 2 depicts our analysis workflow.

Overall, HESTIA features complementary analysis components that operate over the firmware's control-flow graph (CFG) and memory layout. First, the preliminary analysis recovers essential binary metadata, including the firmware base address, construction of the control-flow graph, and reconstruction of the logical `.data` memory region. Second, HESTIA analyzes Heap Management Libraries (HML), locating functions that are provided by the OS or runtime to manage

[1] In the case of NuttX, we assume a non-interactive, production deployment in which application tasks are initialized automatically at system startup rather than being launched via the NSH interactive shell.

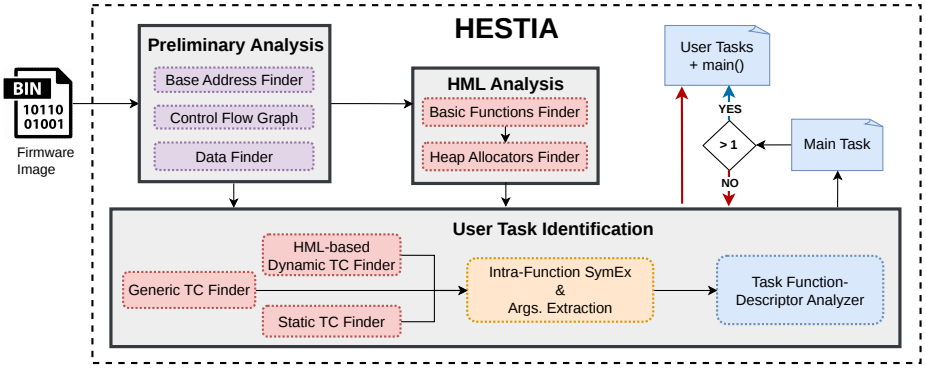

Fig. 2: Overview of HESTIA. Colors indicate analysis components: red denotes task-creation function identification, yellow denotes call-graph traversal and symbolic execution of `TC` call sites for argument extraction, and blue denotes task descriptor analysis used to classify user tasks.

dynamic memory (e.g., `malloc(size)`)—this is useful to then analyze RTOS configurations that rely on dynamic memory allocation. Using the information obtained from the preliminary and HML analyses, HESTIA identifies `TC` functions and task candidates by following the call chain from the firmware's reset handler up to a given depth. Depending on the number of task candidates, the final step of our approach consists of three options:

1. If a single task is identified, we assume that the firmware follows either the Bootstrap-Centric User Task or the Delegated model with the identified task being the main task. In this case, we carry out an additional step to identify further user tasks, by re-applying our analysis to the code of the discovered main task. Finally, we mark as `main` the last function called from a basic block consisting solely of an unconditional branch and taking no arguments.
2. If more than one task candidate is identified, we assume that the RTOS follows the Direct User Task model and consider identified tasks as user tasks. While no additional analysis is required in this case, we further flag the function with the most task-creation calls as the RTOS `main` function.
3. If no task candidate is identified, we conclude that the firmware does not expose explicit task-creation functions within the analyzed call-graph bounds.

This approach leads to a *tiered* analysis model for user-task identification. For RTOSs following Direct User Task model semantics, a single-tier task identification analysis is sufficient. Other initialization models require a two-tiered analysis to overcome indirections introduced by a dedicated main task. Ultimately, HESTIA outputs the set of identified user tasks and their entry points, including the main task and RTOS `main` function when present.

4.1 Preliminary Analysis

Base Address Identification. HESTIA first identifies the correct base address to prevent absolute pointers and vector table entries from referencing incorrect memory regions. Following prior work [46], HESTIA infers the base address by analyzing absolute pointers and evaluating how consistently they resolve to valid code and data regions under different load bases. We select the base address that yields the highest number of consistent pointer-to-target mappings.

Control-Flow Graph Recovery. HESTIA performs static inter-procedural control-flow and function recovery starting from the identified entry point. The analysis lifts basic blocks into an intermediate representation, resolves direct branches, and conservatively approximates indirect control transfers, including register-based branches and function-pointer calls. Function starts are inferred from call targets, architectural prologue patterns, and ARM Cortex-M conventions. The resulting CFG and call graph provide the structural basis for identifying task-creation functions and user tasks.

Data Region Identification. Following the intuition that RTOS firmware code initializes its logical data segments early during startup, HESTIA first inspects all functions reachable from the firmware entry point within a bounded call-graph depth. For these functions, HESTIA inspects instruction semantics to identify the source and destination addresses of memory loads and groups them according to their target memory region and common architecture conventions: addresses within the firmware bounds, as identified by HESTIA under the corrected base address, are treated as Flash-resident, while addresses falling within any of the known SRAM ranges across supported Cortex-M platforms are considered RAM-resident. Among candidate functions, HESTIA selects the one referencing the highest Flash address and at least two distinct RAM addresses as the routine responsible for copying initialized data into RAM. The highest Flash reference indicates the boundary between code and initialized data in Flash. Based on this boundary identification and observed RAM addresses, Hestia establishes the Flash-to-RAM mapping of initialized data, later used during symbolic execution to resolve task descriptor pointers referencing RAM-initialized values.

4.2 Heap Management Library (HML) Analysis

A key observation underlying dynamic task-creation identification is that `TC` functions allocate memory for task descriptors, task stacks, or both, and thus depend on heap allocators (e.g., `malloc`, `zalloc`, etc). In embedded RTOS-based firmware, heap usage is typically reserved for kernel-managed objects such as task descriptors, task stacks, queues, and synchronization primitives, while application logic frequently avoids dynamic allocation due to memory constraints and determinism requirements [8]. Heap-allocator calls are a strong signal for dynamic `TC` identification and help distinguish dynamic `TC` functions from both static `TC` functions and scheduler-related functions that exhibit scheduler-control idioms but do not create tasks.

```
thread_t * thread_create(char *name, thread_start_routine entry,
                         void *arg, int priority, size_t stack_size) {
  thread_t *t = malloc(0x90);     /* descriptor allocation */
  if (!t) return NULL;
  memset(t, 0, 0x90);             /* descriptor init */
  strlcpy(t->name, name, 0x20);   /* task name */
  t->state   = THREAD_SUSPENDED;
  t->priority = priority;
  t->entry   = entry;
  t->arg     = arg;

  void *stk = malloc(stack_size + 0x100); /* stack alloc */
  t->stack = stk;
  if (!stk) { free(t); return NULL; }
  memset(stk, 0x99, 0x100);           /* stack guard */
  memset((uint8_t*)t->stack + 0x100, 0x99, stack_size);
  t->stack_size = stack_size + 0x100;

  disableIRQinterrupts();             /* enter critical section */
  t->thread_list_node.next = thread_list.next;
  t->thread_list_node.prev = &thread_list;
  (thread_list.next)->prev = &t->thread_list_node;
  thread_list.next       = &t->thread_list_node; /* add to task list */
  enableIRQinterrupts();              /* exit critical section */
  return t;
}
```

Fig. 3: Heap-based task descriptor and stack allocation, and scheduler registration in LittleKernel.

Basic Functions Finder. First, HESTIA identifies implementations of fundamental memory manipulation routines, such as `memcpy`- and `memset`-like functions, by analyzing instruction-level memory access behavior. These routines serve as sinks for structured memory initialization and are commonly invoked immediately after memory allocation.

Heap Allocators Finder. Starting from these initialization sinks, HESTIA performs a reaching definition analysis to trace the origins of pointer values whose data flows terminate in the identified memory routines. Functions that generate pointers flowing into such initialization code are classified as heap allocators.

4.3 User Task Identification

Identifying `TC` functions is a prerequisite for recovering user tasks, as `TC` functions initialize tasks within the RTOS scheduler. Depending on the RTOS implementation and configuration, task creation may rely on dynamic allocation, statically allocated task descriptors and stacks, or a combination of both.

Accordingly, HESTIA employs a staged identification strategy comprising two parallel `TC` detection approaches and a fallback: (i) HML-based Dynamic TC detection, which identifies functions exhibiting RTOS scheduler-control idioms that also invoke heap allocators, targeting dynamic task-creation patterns; (ii) Static TC detection, which identifies functions exhibiting the same scheduler-control idioms whose arguments include RAM addresses consistent with statically

allocated task stacks; and (iii) Generic TC detection, a fallback stage activated only when neither (i) nor (ii) yields any candidate, considering all functions based solely on argument patterns to accommodate RTOSs that do not manage tasks as threads (e.g., event-driven RTOSs).

Scheduler-Control Idiom Detection. A key property of `TC` functions is that insertion of a newly created task into scheduler-visible data structures, such as ready lists, is performed inside a critical section or via a kernel-mediated path. HESTIA therefore treats scheduler-control, interrupt-mask-state, and execution-context idioms as a necessary heuristic for identifying task-creation candidates. In practice, our approach recognizes instruction-level patterns including `BASEPRI` interrupt-priority masking, `PRIMASK` save/restore, writes, or mask-state checks, `cpsid i`/`cpsie i` interrupt masking, `IPSR`-based exception-context gates, and `svc`-based supervisor-call gates. These patterns are matched both directly in the candidate function and transitively through bounded-depth callees, covering RTOSs that delegate critical-section entry, interrupt masking, or system-call dispatch to helper routines.

HML Dynamic TC-based Identification. In firmware configurations that support dynamic task creation, task stacks and, in some cases, task descriptors are allocated at runtime via the firmware's HML. As a result, `TC` functions necessarily depend on heap allocators to provision and initialize these memory regions. HESTIA therefore restricts the set of dynamic `TC` candidates to functions within the analyzed call-graph bounds that invoke previously identified heap-allocator functions and exhibit scheduler-control idioms with arguments consistent with task descriptor structures (i.e., function pointers, name buffers, or priority fields).

Figure 3 illustrates a simplified excerpt from LittleKernel's `thread_create` function, showing a common heap-based task-creation pattern in which a dynamically allocated task descriptor and stack are immediately initialized using memory manipulation routines.

Static TC-based Identification. Statically allocated tasks are initialized without invoking the HML and therefore cannot be detected using HML-based filtering. To handle such cases, HESTIA restricts candidates to functions within the analyzed call-graph bounds and applies the same task-creation heuristics used in the HML-based stage, with the additional requirement that argument values include RAM addresses consistent with statically allocated task stacks.

Generic TC-based Identification. When neither HML-based nor Static TC-based identification yields any `TC` candidate, HESTIA falls back to a final, more permissive analysis that considers all functions within the analyzed call-graph bounds and relies solely on the function-argument patterns. While this inevitably introduces false positives, it is needed to handle RTOSs that do not manage tasks as threads but instead invoke tasks as regular functions, such as Contiki—the only RTOS in our dataset following this design.

Symbolic Execution and Argument Extraction. HESTIA employs intra-procedural symbolic execution in a call-less mode (i.e., call instructions are not followed) to resolve and reason about values passed to candidate task-creation

functions. Specifically, for each analyzed call site, HESTIA symbolically executes the enclosing function from its entry point up to the call instruction to recover argument values according to the ARM calling convention implemented by the underlying analysis framework. This includes both register arguments (`r0-r3`) and potential stack-passed arguments. To capture the latter, HESTIA inspects stack-related memory operations nearby the call (e.g., `ldr`/`str` via `sp`, as well as `ldm`/`stm`) and records values written to or loaded from stack argument slots.

If the call site cannot be reached through structured intra-procedural traversal from the function entry (e.g., due to imprecise control-flow recovery), HESTIA initializes symbolic execution directly at the call site and relies on the enclosing basic block to extract both register and stack arguments using the same stack- and memory-access inspection. Recovered values are treated as candidate pointers and checked to determine whether they reference valid functions or memory regions consistent with task descriptor data structures.

Symbolic execution is bounded to individual functions and limited call-graph depth, ensuring scalability while preserving sufficient context to resolve task-related arguments.

Task Function-Descriptor Analysis. For each argument recovered via symbolic execution, HESTIA determines whether it corresponds to a task entry function pointer or to a pointer to a task descriptor structure by analyzing the resolved argument value. Because task-creation APIs differ across RTOS implementations, arguments passed to task-creation functions may directly contain a task entry function or a pointer to a task descriptor structure.

If an argument resolves directly to a function address (or to a pointer that resolves to a function address), HESTIA treats it as a candidate task entry function. In this case, HESTIA searches the remaining arguments of the same call site for a pointer to a valid ASCII string, which is interpreted as the task name. If such a string is found, the function is classified as the entry point of a user task with an associated name, part of the task descriptor; otherwise, it is classified as a nameless user task.

If an argument does not resolve to a function address and may instead represent a pointer to a task descriptor structure, HESTIA attempts to resolve it by inspecting memory at the referenced address. Specifically, HESTIA analyzes up to 96 bytes of memory (12 contiguous 32-bit entries) starting at the pointed address. This bound is derived from empirical observation: among the RTOSs considered, Zephyr uses the largest task descriptor, comprising 12 entries.

The inspected memory region is analyzed incrementally, entry by entry. The first entry must be directly referenced by the task-creation call site, indicating that the argument points to the beginning of a structured object. Subsequent entries are examined until either the required descriptor fields are identified or a termination condition is reached. The analysis terminates unsuccessfully if two consecutive entries resolve to valid function pointers or two consecutive entries resolve to valid ASCII strings, as such patterns are indicative of unrelated global data rather than a contiguous task descriptor structure. Additionally, apart from the first entry, none of the inspected entries may be statically referenced elsewhere

in the firmware. If any subsequent entry is referenced by other instructions, the region is treated as independent global data rather than a task descriptor, and the analysis terminates early.

A region is classified as a task descriptor if, within the inspected entries prior to termination, HESTIA identifies both (i) a pointer to a valid task entry function and (ii) a pointer to a valid ASCII string, interpreted as the task name.

For a function address to be considered a valid task entry function, it must satisfy additional constraints. Specifically, (1) the function must be present in the set of functions identified during control-flow graph construction, (2) have no predecessors in the call graph (indicating that it is not invoked through regular control flow), (3) contain a minimum amount of executable code (to exclude empty or trivial stubs), and (4) not contain architecture-specific synchronization or barrier instructions (e.g., `isb`, `dsb`, or `dmb`), which are indicative of scheduler or low-level system routines rather than application-level task handlers.

Finally, HESTIA attempts to identify the overall `main` function of the firmware. For RTOSs using the Direct User Task Model, we identify `main` by locating the function initializing most of the user tasks before entering a loop. For firmware following the Bootstrap-Centric User Task Model—where `main` may be invoked within the main task—HESTIA determines `main` as the last function called from a basic block consisting solely of an unconditional branch and taking no arguments. This heuristic matches the execution model for all analyzed RTOSs under this model, except NuttX, where the main task is fully developer-controlled.

Using these criteria, HESTIA classifies the output into three categories: full task descriptors when the task has an associated name, user-task entry functions for nameless tasks, and a `main` function.

5 Implementation

We implement HESTIA in Python on top of the analysis framework `angr` [40]. `angr` provides a static and symbolic analysis infrastructure for binary lifting, which we leverage to implement the analysis components described in Section 4.

Preliminary Analysis. HESTIA recovers the firmware base address using a technique inspired by FirmXRay [46] and configures the `angr` project accordingly. The firmware entry point (i.e., the Reset Handler) is extracted from the vector table and used to initialize analysis. Then, HESTIA constructs a control-flow graph using `angr`'s `CFGFast` analysis to recover function boundaries and call relationships. For data region identification, HESTIA uses an initial call-depth of three to identify initialization functions and classifies candidate addresses using the recovered firmware bounds and known SRAM ranges across supported Cortex-M platforms [30].

Task Creation Function Analysis. HESTIA identifies task-creation functions using the staged analysis described in Section 4. Dynamic task-creation functions are detected via an HML analysis inspired by Heapster [22], by recovering heap allocators and selecting allocator-dependent functions that exhibit scheduler-control idioms and task-argument patterns. Static task-creation functions use

the same idioms and argument patterns, but rely on SRAM addresses consistent with statically allocated task descriptors and stacks. If neither stage yields a candidate, HESTIA falls back to generic argument-pattern analysis.

Application Code Identification. Given the identified task-creation functions, HESTIA uses bounded symbolic execution to recover their arguments, then analyzes these arguments to identify task descriptors, task entry functions, task names when present, and the `main` function when applicable, as described in Section 4.

6 Evaluation

We evaluate HESTIA on three datasets comprising 145 firmware binaries. We run all our experiments on an Intel(R) Core(TM) Ultra 7 155H server (16 cores, 22 threads, 4.8 GHz) with 32 GB of RAM, running Ubuntu 22.04 LTS.

6.1 Datasets

Baseline. Our baseline dataset contains 51 RTOS-based firmware images spanning RIOT, Zephyr, Mbed OS, Contiki-NG, LittleKernel, NuttX, and FreeRTOS. It combines firmware built from official RTOS repositories with real-world firmware from the `ucsb-seclab` monolithic firmware collection[2], which consists of binaries analyzed in prior work [17,10,26,41,24,34,33,13,49,42,32]. We manually build 25 images from official RTOS repositories using GCC: 5 at `-O0`, 3 at `-O3`, and 17 at `-Os`. The remaining 26 RTOS-based images from the collection consist of 2 images compiled at `-O0`, 2 at `-O1`, and 22 at `-Os`; we recover these optimization levels from DWARF debug information present in the ELF files. The dataset covers multiple Cortex-M platforms across various vendors and application domains, including device control, networking services, IoT nodes, and embedded command shells. For each firmware image, we use the ELF file with symbols as ground truth for evaluation, which we then strip and convert into a raw binary as found on real-world devices using `arm-none-eabi-strip` and `arm-none-eabi-objcopy`.

RTOSExtracter. To assess the capabilities of HESTIA when applied to a dataset containing previously unseen RTOSs, we also evaluate on a subset of RTOSExtracter's dataset [47].[3] This dataset comprises 14 open-source smart IoT projects from GitHub, covering one known (FreeRTOS) and four unseen RTOSs (LiteOS, RT-Thread, μC/OS-II, μC/OS-III), compiled at different optimization levels using three different toolchains (Keil, IAR, and GCC). This corresponds to 44 firmware binaries compiled with the three toolchains and two optimization levels (`-O0` and `-O3`). The distribution of the obtained binaries is as follows: 16

[2] https://github.com/ucsb-seclab/monolithic-firmware-collection

[3] Both dataset and its ground truth of RTOSExtracter are not publicly available. Upon request, we obtained a subset of the paper's original dataset as raw binaries from the authors and re-constructed ground truth, where possible, from the provided artefact.

images compiled with Keil, 14 with GCC, and 14 with IAR; 22 with optimization level `-O0` and 22 with `-O3`.

True Negative. Last, to understand the robustness of HESTIA when confronted with firmware not exhibiting any RTOS, we leverage a dataset of solely bare-metal firmware. For this, we collect 50 additional firmware samples from the `ucsb-seclab` monolithic firmware collection, and we manually verify that they do not include any RTOS.

6.2 Identification Performance

First, we evaluate the effectiveness of HESTIA in identifying user tasks on our baseline dataset. We use three primary metrics: (i) *task recall*, defined as the fraction of ground-truth user tasks correctly identified, (ii) *task precision*, defined as the fraction of identified user-task candidates that correspond to true user tasks; and (iii) *search-space reduction rate*, defined as the reduction in the number of functions requiring manual inspection to identify user tasks. We derive ground-truth user tasks by analyzing all created tasks, excluding system tasks found in RTOS source code, shell-only tasks, and task descriptors present in flash but not referenced, initialized, or scheduled at runtime; we also include `main` functions.

We conduct the evaluation under multiple call-graph bounding configurations, where m and n denote the exclusive exploration depth from the firmware entry point and the main task, respectively. Specifically, we consider $(m, n) \in \{(3,2), (3,3), (3,4), (4,2), (4,3), (4,4)\}$, with results summarized in Table 1.

Across the 51 firmware images, HESTIA identifies 130 of 131 ground-truth user tasks, achieving a recall of 99.24%, with the highest recall obtained under the configuration $m = 3, n = 3$. A per-RTOS breakdown for this configuration is provided in Table 3 (see Appendix A).

When $n = 2$, HESTIA misses tasks initialized deeper in the call graph originating from the main task. Increasing n to 3 recovers these tasks, while increasing n beyond 3 does not further improve recall. The remaining false negative is the `main` function in one FreeRTOS sample, where the firmware exhibits Delegated-model characteristics despite generally following the Direct User Task model, leading to incorrect initialization-model attribution.

Increasing m, in contrast, primarily affects precision. Larger exploration depth leads to more false positives, especially in firmware following the Direct User Task model (i.e., Contiki-NG and FreeRTOS), while not yielding any recall improvements. In principle, in firmware following the Bootstrap-Centric User Task and Delegated models, false positives in identifying the main task could cause incorrect model selection and, consequently, missed user tasks, although this behavior was not observed in our experiments.

Approximately 50% of the false positives originate from Contiki-based images. Contiki follows an event-driven execution model and does not manage tasks as preemptive threads; instead, execution units are invoked as regular functions that cooperatively yield control. When explicit `TC` functions cannot be identified, HESTIA falls back to the Generic TC-based analysis described in Section 4.3.

Table 1: HESTIA evaluation on the baseline dataset. Here, m and n bound the exclusive call-graph depth (starting at depth 0) explored from the firmware entry point and the main task respectively.

Parameter	Found/True Tasks	False Positives	Precision (%)	Recall (%)	Search-Space Reduction (%)†
$m = 3, n = 2$	129/131	38	77.25	98.47	99.58
$m = 3, n = 3$	130/131	41	76.02	99.24	99.57
$m = 3, n = 4$	130/131	42	75.58	99.24	99.57
$m = 4, n = 2$	128/131	43	74.85	97.71	99.57
$m = 4, n = 3$	129/131	45	74.14	98.47	99.56
$m = 4, n = 4$	129/131	47	73.30	98.47	99.56

† Percentage reduction from all functions in the firmware binary to the shortlist of potential application code entry points computed as SSR $= \frac{F_{\text{all}} - T_{\text{cand}}}{F_{\text{all}}} \times 100\%$.

This fallback relies solely on function-argument patterns and structural cues, which are insufficient to distinguish task entry functions from callback handlers or virtual table targets. As a result, functions passed to non-task-related APIs may be misclassified as user tasks.

Zephyr-based firmware accounts for the $\approx 25\%$ portion of the false positives. Although `TC` functions are reliably identified in Zephyr-based images, Zephyr initializes both user and system tasks within shared initialization routines, making them structurally indistinguishable under our current abstraction.

Additionally, the presence of multiple function calls within the main task leads to false positives. Following our heuristics, HESTIA conservatively marks all of them as potential `main` functions. The remaining false positives stem from unnamed system task initialization in FreeRTOS, as well as signal-handling APIs (e.g., `sigaction`-like mechanisms) in NuttX, whose callback-driven execution paths appear similar to those employed in task instantiation.

Despite these challenges, HESTIA substantially reduces the analyst's search space. Considering all functions in a firmware binary as initial candidates, HESTIA achieves an average function search-space reduction rate of 99.5%, meaning that fewer than 1% of functions require (manual) inspection.

6.3 Runtime Performance

We evaluate the runtime performance of HESTIA by reporting end-to-end analysis times on the baseline dataset, with detailed results provided in Table 4 (see Appendix A). On average, HESTIA completes analysis in $\approx$ 15 minutes per firmware image. The dominant cost stems from the HML analysis, an inherently expensive data-flow pipeline. In contrast, the User Task Identification stage is lightweight, as it operates on bounded symbolic execution and static-analysis-based heuristics, completing in $\approx$ 34 seconds on average.

6.4 Identification Performance against Unseen RTOSs

Table 2 presents the results of the evaluation on the RTOSExtracter dataset [47]. We run HESTIA with $n = 3$ and m ranging from 2 to 4 and report the configuration that identifies the first task. For 21 samples out of the 30 compiled with Keil and IAR, we manually specify the first function reachable from the firmware entry point, as `angr` was unable to correctly construct the call graph. Compared with the baseline dataset, where Hestia achieves 99.24% recall under the best configuration, recall drops to 78.05% on the RTOSExtracter dataset.

For FreeRTOS ($m = 3$) and LiteOS ($m = 4$), both generally following the Direct User Task model, several samples exhibit Delegated-model characteristics. This is expected for FreeRTOS, which is provided as a library and gives developers flexibility in task creation and structure. HESTIA still recovers all FreeRTOS tasks, but misses `main` due to incorrect initialization-model attribution. For LiteOS, `OSAppInit` is missing from `OSMain` in all but two images, leaving task descriptors present but unreachable from code. In the two remaining images, the Delegated model structure again prevents HESTIA from recovering `main`.

RT-Thread ($m = 4$) follows the Bootstrap-Centric User Task model. HESTIA achieves perfect precision and misses only one task. This false negative stems from `angr` failing to recover the corresponding `main` routine as a function, preventing HESTIA from identifying it.

μC/OS-II ($m = 2$) and μC/OS-III ($m = 3$) follow the Direct User Task model. However, similar to the FreeRTOS samples, some μC/OS-II samples resemble the Delegated model, due to the RTOS's flexibility in main task initialization. This task is sometimes initialized alongside system tasks, such as the Idle and Timer tasks, and μC/OS-II does not include task names during initialization. In these cases, HESTIA terminates at the first tier and misses tasks initialized within the main task. In contrast, μC/OS-III includes task names, which HESTIA leverages, resulting in higher recall.

Finally, Table 2 shows that HESTIA performs worse on binaries compiled with `-O3`. The primary reason is that aggressive compiler optimizations (e.g., inlining, interprocedural analysis, code motion) flatten the call graph and obscure the hierarchical separation between system and user-task initialization.

6.5 Robustness under Different Firmware Types

Analysts may not be initially aware of whether a given monolithic firmware image deploys an RTOS or runs in a bare-metal (i.e., Type-III) configuration, without any OS abstractions. This poses a significant challenge for our approach, as such firmware images do not contain user tasks. False-positive detection of tasks may undermine automated security analysis or impede manual reverse engineering.

When applied to the true negative dataset under a $m = 3$ and $n = 3$ configuration, HESTIA falsely identifies a total of 33 tasks in 14 out of the 50 firmware samples, leading to a false-positive rate of 28%. Yet, recent research showed promising results in automatically distinguishing RTOS-based from bare-metal firmware [35].

Table 2: HESTIA evaluation on the RTOSExtracter dataset. (**F: Found**, **T: True**, **Rec: Recall**, **Prec: Precision**)

RTOS	**Total**				**O0**			**O3**		
	# Samples	**F/T Tasks**	**Rec. (%)**	**Prec. (%)**	**F/T Tasks**	**Rec. (%)**	**Prec. (%)**	**F/T Tasks**	**Rec. (%)**	**Prec. (%)**
FreeRTOS	4	16/20	80.00	94.12	8/10	80.00	100.0	8/10	80.00	88.89
LiteOS	12	2/6	33.33	25.00	2/3	66.67	25.00	0/3	0.0	0.0
RT-Thr.	10	19/20	95.00	100.0	10/10	100.0	100.0	9/10	90.00	100.0
μCOS-II	12	61/80	76.25	98.39	32/40	80.00	96.97	29/40	72.50	100.0
μCOS-III	6	30/38	78.95	90.91	15/19	78.95	93.75	15/19	78.95	88.24
Overall	**44**	**128/164**	**78.05**	**92.09**	**67/82**	**81.71**	**89.33**	**61/82**	**74.39**	**95.31**

7 Discussion

Call-graph Exploration. The key parameter for HESTIA is the chosen call-graph depth. Increasing the depth enables the recovery of user tasks that reside farther along the same initialization layer (i.e., deeper from the main task/function). However, broader exploration may increase false positives by incorporating unrelated application logic, without substantially improving recall. This effect is particularly visible in RTOS-based firmware that does not employ thread-based task management (e.g., in Contiki-NG which uses an event-driven execution model). In such cases, no explicit `TC` functions are identified and HESTIA falls back to the generic `TC`-based analysis, providing a trade-off between acceptable false-positive rates with the generality to scale to a wide range of RTOSs. Moreover, aggressive compiler optimizations tend to flatten the call graph, further exacerbating this trade-off.

Two-Tiered Analysis. Our tiered analysis approach addresses a different structural concern. The two-tier analysis helps isolate the main task in the Bootstrap-Centric User Task and Delegated models from most system tasks, which are typically spawned along deeper kernel initialization paths or created via indirect, data-driven mechanisms. This separation is important, as HESTIA does not semantically distinguish between system-managed and user tasks.

While system-task initialization within user tasks is rare under the two-tier setting, it occurs more frequently in the one-tier case (i.e., under the Direct User Task model), where system and user tasks may be initialized together and thus introduce additional false positives.

On the other hand, for samples under the Delegated and Bootstrap-Centric models, any false positive in the first analysis tier causes incorrect model selection, preventing HESTIA from reaching the second tier and thereby missing user tasks initialized in the main task. While HESTIA identifies common system tasks by name (e.g., Idle, Timer, Tick) to limit false positives, we leave improved identification and filtering for future work.

Finally, in both one-tier and two-tier analysis, user tasks may also be missed when they initialize additional user sub-tasks, since recovering such cases would require an additional analysis tier; however, we observe this pattern to be rare in practice.

At the same time, the tiered design is essential for bridging system-level initialization and user-task creation. Thus, we rely on a deliberate trade-off between coverage and precision grounded in empirically observed firmware structure.

Main Function Identification. Evaluation results indicate that `main` function identification in the Bootstrap-Centric model introduces false positives due to the presence of multiple function calls within the main task that resemble `main` function invocations. Conversely, Direct User Task samples that resemble the Delegated model are misclassified by Hestia, resulting in false negatives, as Hestia incorrectly assumes that `main` resides within the main task. Future work could analyze call-graph structure around `main`, including scheduler initialization patterns, to better distinguish Direct User Task from other initialization models.

Hestia and Bare-metal Firmware. While our evaluation showcases the limitations of Hestia on bare-metal firmware, surveys indicate that over 70% of embedded projects are using an operating system with a majority relying on FreeRTOS [5,15], demonstrating the relevancy of Hestia for most analysis tasks. Additionally, future work can improve Hestia's accuracy by combining our approach with existing work on RTOS identification [35] as an additional analysis step, ultimately reducing the false positives encountered by analyzing non-RTOS firmware images.

Limitations. As with any static approach, Hestia inherits fundamental limitations deriving from static analysis, such as imprecise CFGs, approximated indirect jumps, and missing call edges. Following this, our results indicate that higher compiler optimization levels (i.e., `-O3`) and different compilers (Keil and IAR) introduce additional challenges for Hestia. Future work should further evaluate the generalizability of the proposed heuristics under the `-O3` optimization level, as well as improve call-graph recovery from firmware entry points to reduce reliance on manual specification.

8 Related work

RTOSExtracter. Closest to our work, RTOSExtracter [47] identifies task code in RTOS firmware; however, its approach differs substantially from ours. A direct comparison with RTOSExtracter was not possible due to unavailable artifacts, reliance on non-reproducible legacy dependencies, and incomplete access to the original evaluation dataset required for a sound comparison. RTOSExtracter requires the corresponding RTOS SDK to identify task-creation APIs and manually collected parameter-structure prototypes for each supported compiler. Although recent work has shown promising results in automatically identifying RTOS frameworks in embedded firmware [35], in practice, obtaining the exact SDK version that was used to compile the target firmware is challenging. This

challenge is exacerbated by the lack of metadata in firmware blobs regarding the RTOS, SDK version, or compiler used. While RTOSExtracter demonstrates strong accuracy and efficient runtime performance when such information is available, collecting this metadata in real-world scenarios remains challenging.

Conceptually, RTOSExtracter defines user-defined functions operationally as task entry functions passed to RTOS task-creation APIs. As a result, it collects all tasks instantiated through these APIs without distinguishing between application-level and system-level tasks. Since the same APIs are typically used for both, application and kernel tasks are conflated. In contrast, Hestia distinguishes between these categories and additionally recovers the `main` function.

Code Identification for Monolithic Firmware. Only a limited amount of work has attempted to separate system-level code from developer-defined application code in monolithic firmware. HALucinator [10] matches identified firmware functions to known HAL functions in a database. While this approach allows to reduce the search space for application code, it requires a priori knowledge of known HAL functions and does not identify custom, user tasks in RTOSs.

SFuzz [9] identifies applications by tracking call graphs starting from functions that handle user inputs, aiming to steer fuzzing efforts. They depend on binary and heuristic matching to identify functions that handle user inputs. Tasks that are internally triggered or not exposed to external inputs are excluded.

Code Identification For Linux-based Firmware. While Linux-based firmware provides a clear separation between OS kernel and userspace binaries, identifying programs introduced by firmware developers faces similar challenges to our work. Thus, approaches for code identification and security analysis typically assume that the specific target application or its application domain is already known: Costin et al [12] and Gao et al [20] perform analysis starting from web server binaries, Greenhouse [45] rehosts analyst-specified applications, and CryptoRex [48] identifies misuse of cryptographic libraries. In contrast, Karonte [39] analyzes the interaction between multiple applications and Operation Mango [21] performs analysis of all applications affecting the firmware environment.

While all of these works focus on Linux-based systems, they demonstrate that effectively distinguishing developer-introduced code from the broader firmware ecosystem is a crucial necessity for effective security analysis—a requirement addressed by Hestia for RTOS firmware.

9 Conclusions

We presented Hestia, a new analysis tool that automatically identifies application entry points in RTOS firmware images. We evaluated our tool on 145 firmware samples and demonstrated that Hestia consistently reduces the number of functions to be (manually) inspected by 99%. Ultimately, our tool allows security analyses to focus on *what matters*, enabling micro-patching and directed fuzzing, and guiding static analyses to relevant program regions, bounding expensive exploration.

Acknowledgments. We would like to thank our reviewers and shepherd for their valuable inputs to improve our paper. This work has been partially supported by the "INSYST" TKI project, by the "HIVA" project of the Vidi TTW research programme [No. 22249, grant ID https://doi.org/10.61686/LMSYB97788] funded by the Dutch Research Council (NWO), and by the Dutch Ministry of Economic Affairs and Climate Policy (EZK) through the AVR project "FirmPatch". Peng Liu was supported by the China Scholarship Council (CSC) scholarship, 2023.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

A Hestia Per-RTOS Evaluation and Runtime Results

Table 3: Hestia per-RTOS evaluation on the baseline dataset using the optimal parameter configuration $m = 3, n = 3$.

RTOS	# Samples	Found/True Tasks	False Positives	Precision (%)	Recall (%)	Search-Space Reduction (%)†
RIOT	7	15/15	1	93.75	100.00	99.70
LittleKernel	3	9/9	0	100.00	100.00	99.52
Zephyr	19	60/60	10	85.71	100.00	99.51
Mbed OS	3	6/6	0	100.00	100.00	99.76
Contiki-NG	11	22/22	26	45.83	100.00	99.36
NuttX	5	7/7	2	77.78	100.00	99.81
FreeRTOS	3	11/12	2	84.62	91.67	99.58
Overall	**51**	**130/131**	**41**	**76.02**	**99.24**	**99.57**

† Percentage reduction from all functions in the firmware binary to the shortlist of potential application code entry points computed as SSR $= \frac{F_{\text{all}} - T_{\text{cand}}}{F_{\text{all}}} \times 100\%$.

Table 4: Hestia runtime analysis on the baseline dataset using the optimal parameter configuration $m = 3, n = 3$. We report per-stage timing (average, median) and total execution time (average, median, minimum, maximum) per RTOS; all times are in `mm:ss` format. The Pre-Analysis and HML stages are aggregated since the former has negligible runtime impact.

RTOS	#Samples	Avg. Func.	Pre & HML		User Tasks		Total Exec. Time			
			Avg.	Median	Avg.	Median	Avg.	Median	Min	Max
RIOT	7	760	19:08	06:54	00:31	00:21	19:48	07:17	00:57	62:17
LittleKernel	3	626	09:32	10:10	01:01	01:04	10:41	11:06	06:00	14:56
Zephyr	19	758	13:02	13:33	00:19	00:17	13:28	14:12	05:05	17:18
Mbed OS	3	832	08:18	01:10	00:09	00:06	08:39	01:23	01:21	23:14
Contiki-NG	11	687	19:39	19:47	00:47	00:23	20:41	20:26	14:44	25:53
NuttX	5	968	14:14	14:03	00:58	01:04	15:24	14:49	07:32	27:16
FreeRTOS	3	1030	17:32	17:10	00:45	00:37	18:40	17:51	11:31	26:37
Overall	**51**	**776**	**15:12**	**15:49**	**00:34**	**00:20**	**15:56**	**16:59**	**00:57**	**62:17**

B RTOS Task Initialization Patterns

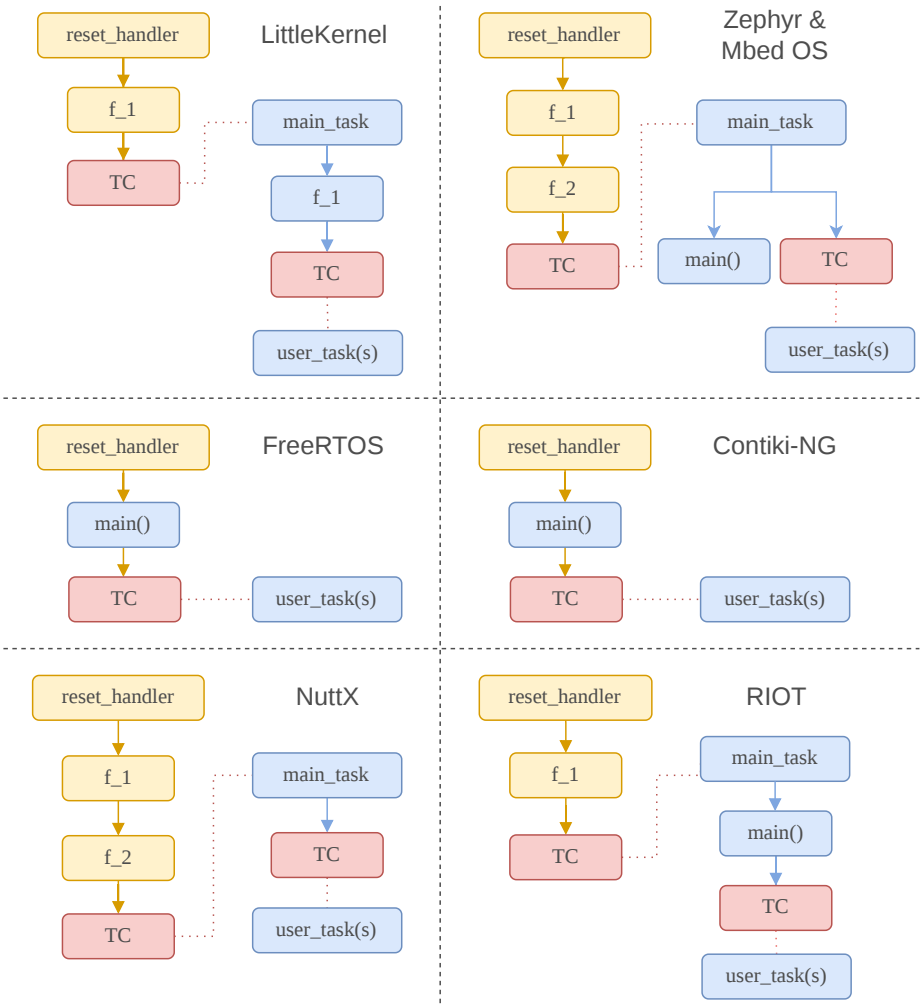

Fig. 4: Schematic overview of concrete RTOS task initialization patterns. Here, `f_n` denotes call-graph levels from the entry point or main task and `TC` marks task-creation APIs. Solid arrows (→) indicate direct function calls, while dotted arrows (⋯) indicate task creation. Yellow highlights the function call chain leading to the first initialized task, red denotes `TC` functions, and blue denotes application code.

References

1. Abbasi, A., Wetzels, J., Holz, T., Etalle, S.: Challenges in designing exploit mitigations for deeply embedded systems. In: Procs. of the IEEE European Symposium on Security and Privacy (EuroS&P) (2019)
2. All, I.F.: 5 Worst IoT Hacking Vulnerabilities (2026), https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities, accessed: 2026-02-04
3. Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., Zhou, Y.: Understanding the Mirai Botnet. In: Procs. of the USENIX Security Symposium (2017)
4. Arm: Mbed OS, https://github.com/ARMmbed/mbed-os
5. Aspencore: The Current State of Embedded Development (2023)
6. Balgavy, A., Muench, M.: Firmline: a generic pipeline for large-scale analysis of non-linux firmware. In: Procs. of the Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research (BAR) (2024)
7. Baran, G.: Massive IoT Data Breach Exposes 2.7 Billion Records, Including Wi-Fi Passwords (2025), https://cybersecuritynews.com/massive-iot-data-breach/, accessed: 2026-02-04
8. Buttazzo, G.C.: Hard real-time computing systems: predictable scheduling algorithms and applications. Springer (1997)
9. Chen, L., Cai, Q., Ma, Z., Wang, Y., Hu, H., Shen, M., Liu, Y., Guo, S., Duan, H., Jiang, K., Xue, Z.: SFuzz: Slice-based Fuzzing for Real-Time Operating Systems. In: Procs. of the ACM Conference on Computer and Communications Security (CCS) (2022)
10. Clements, A.A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., Payer, M.: HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In: Procs. of the USENIX Security Symposium (2020)
11. Contiki-NG: Contiki-NG: The OS for Next Generation IoT Devices, https://github.com/contiki-ng/contiki-ng
12. Costin, A., Zarras, A., Francillon, A.: Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In: Procs. of the ACM ASIA Conference on Computer and Communications Security (ASIACCS) (2016)
13. Duong, M., Chesser, M., Farrelly, G., Nepal, S., Ranasinghe, D.C.: FirmReBugger: A Benchmark Framework for Monolithic Firmware Fuzzers. In: Procs. of the USENIX Security Symposium (2026)
14. Ebbers, F.: A large-scale analysis of iot firmware version distribution in the wild. IEEE Transactions on Software Engineering **49**(2) (2022)
15. Emilio, M.D.P.: Embedded Systems Survey 2025: Insights and Trends (2025)
16. Fasano, A., Ballo, T., Muench, M., Leek, T., Bulekov, A., Dolan-Gavitt, B., Egele, M., Francillon, A., Lu, L., Gregory, N., et al.: Sok: Enabling Security Analyses of Embedded Systems via Rehosting. In: Procs. of the ACM ASIA Conference on Computer and Communications Security (ASIACCS) (2021)
17. Feng, B., Mera, A., Lu, L.: P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In: Procs. of the USENIX Security Symposium (2020)
18. Foundation, A.S.: NuttX, https://github.com/apache/nuttx
19. FreeRTOS.Org: FreeRTOS(TM) is a market leading RTOS from Amazon Web Services, https://github.com/FreeRTOS

20. Gao, Z., Zhang, C., Liu, H., Sun, W., Tang, Z., Jiang, L., Chen, J., Xie, Y.: Faster and Better: Detecting Vulnerabilities in Linux-based IoT Firmware with Optimized Reaching Definition Analysis. In: Procs. of the Symposium on Network and Distributed System Security (NDSS) (2024)
21. Gibbs, W., Raj, A.S., Vadayath, J.M., Tay, H.J., Miller, J., Ajayan, A., Basque, Z.L., Dutcher, A., Dong, F., Maso, X., et al.: Operation mango: Scalable discovery of Taint-Style vulnerabilities in binary firmware services. In: Procs. of the USENIX Security Symposium (2024)
22. Gritti, F., Pagani, F., Grishchenko, I., Dresel, L., Redini, N., Kruegel, C., Vigna, G.: HEAPSTER: Analyzing the Security of Dynamic Allocators for Monolithic Firmware Images. In: Procs. of the IEEE Symposium on Security and Privacy (S&P) (2022)
23. Gustafson, E., Grosen, P., Redini, N., Jha, S., Continella, A., Wang, R., Fu, K., Rampazzi, S., Kruegel, C., Vigna, G.: Shimware: Toward practical security retrofitting for monolithic firmware images. In: Procs. of the Symposium on Recent Advances in Intrusion Detection (RAID) (2023)
24. Gustafson, E., Muench, M., Spensky, C., , Redini, N., Machiry, A., Francillon, A., Balzarotti, D., Choe, Y.R., Kruegel, C., Vigna, G.: Toward the Analysis of Embedded Firmware Through Automated Re-hosting. In: Procs. of the Symposium on Recent Advances in Intrusion Detection (RAID) (2019)
25. Herwig, S., Harvey, K., Hughey, G., Roberts, R., Levin, D.: Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet. In: Procs. of the Symposium on Network and Distributed System Security (NDSS) (2019)
26. Hofhammer, F., Busch, M., Wang, Q., Egele, M., Payer, M.: SURGEON: Performant, Flexible, and Accurate Re-Hosting via Transplantation. In: Procs. of the Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research (BAR) (2024)
27. Insights, T., Topics, E.: Number of Internet of Things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033 (in billions) (2024), https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/, online
28. Kernel, L.: The Little Kernel Embedded Operating System, https://github.com/littlekernel/lk
29. Laplante, P.A., Rose, E.P., Gracia-Watson, M.: An historical survey of early real-time computing developments in the U.S. Real-Time Systems **8**(2) (1995)
30. Ltd., A.: ARMv7-M Architecture Reference Manual. ARM (2010), section B3: Memory model and address map
31. Luna, R., Islam, S.A.: Security and Reliability of Safety-Critical RTOS. SN Computer Science **2**(5) (2021)
32. Maklad, Y., Wael, F., Hamdi, A., Elsersy, W., Shaban, K.: MultiFuzz: A Dense Retrieval-based Multi-Agent System for Network Protocol Fuzzing. In: Procs. of the ACM ASIA Conference on Computer and Communications Security (ASIACCS) (2025)
33. Mera, A., Feng, B., Lu, L., Kirda, E.: DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In: Procs. of the IEEE Symposium on Security and Privacy (S&P) (2021)
34. Muench, M., Stijohann, J., Kargl, F., Francillon, A., Balzarotti, D.: What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In: Procs. of the Symposium on Network and Distributed System Security (NDSS) (2018)

35. van Nielen, J., Peter, A., Continella, A.: frameD: Toward automated identification of embedded frameworks in firmware images. In: Procs. of the European Symposium on Research in Computer Security (ESORICS) (2024)
36. Nino, N., Lu, R., Zhou, W., Lee, K.H., Zhao, Z., Guan, L.: Unveiling IoT security in reality: A Firmware-Centric journey. In: Procs. of the USENIX Security Symposium (2024)
37. OS, R.: RIOT OS: The friendly Operating System for IoT, https://github.com/RIOT-OS/RIOT
38. Project, T.Z.: Zephyr Project, https://github.com/zephyrproject-rtos/zephyr
39. Redini, N., Machiry, A., Wang, R., Spensky, C., Continella, A., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Karonte: Detecting insecure multi-binary interactions in embedded firmware. In: Procs. of the IEEE Symposium on Security and Privacy (S&P) (2020)
40. Salwan, J., Shoshitaishvili, Y., et al.: angr: A Multi-Architecture Binary Analysis Platform. In: Proceedings of the IEEE Symposium on Security and Privacy Workshops (SPW) (2016), https://angr.io
41. Scharnowski, T., Bars, N., Schloegel, M., Gustafson, E., Muench, M., Vigna, G., Kruegel, C., Holz, T., Abbasi, A.: Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In: Procs. of the USENIX Security Symposium (2022)
42. Scharnowski, T., Wörner, S., Buchmann, F., Bars, N., Schloegel, M., Holz, T.: HOEDUR: Embedded Firmware Fuzzing using Multi-stream Inputs. In: Procs. of the USENIX Security Symposium (2023)
43. Seri, B., Vishnepolsky, G., Zusman, D.: Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS. Tech. rep., Armis (2019), https://info.armis.com/rs/645-PDC-047/images/Urgent11%20Technical%20White%20Paper.pdf
44. Tan, X., Ma, Z., Pinto, S., Guan, L., Zhang, N., Xu, J., Lin, Z., Hu, H., Zhao, Z.: SoK: Where's the "up"?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems. In: Procs. of the USENIX Workshop on Offensive Technologies (WOOT) (2024)
45. Tay, H.J., Zeng, K., Vadayath, J.M., Raj, A.S., Dutcher, A., Reddy, T., Gibbs, W., Basque, Z.L., Dong, F., Smith, Z., et al.: Greenhouse: single-service rehosting of linux-based firmware binaries in user-space emulation. In: Procs. of the USENIX Security Symposium (2023)
46. Wen, H., Lin, Z., Zhang, Y.: FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware. In: Procs. of the ACM Conference on Computer and Communications Security (CCS) (2020)
47. Xie, X., Ye, J., Wu, L., Li, R.: RTOSExtracter: Extracting user-defined functions in stripped RTOS-based firmware. In: International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC). IEEE Computer Society (2022)
48. Zhang, L., Chen, J., Diao, W., Guo, S., Weng, J., Zhang, K.: CryptoREX: Large-scale analysis of cryptographic misuse in IoT devices. In: Procs. of the Symposium on Recent Advances in Intrusion Detection (RAID) (2019)
49. Zhou, W., Guan, L., Liu, P., Zhang, Y.: Automatic Firmware Emulation through Invalidity-guided Knowledge Inference (Extended Version). In: Procs. of the USENIX Security Symposium (2021)