RTCon: Context-Adaptive Function-Level Fuzzing for RTOS Kernels

Eunkyu Lee
KAIST
School of
Electrical Engineering
ekleezg@kaist.ac.kr

Junyoung Park
KAIST
School of
Electrical Engineering
parkjuny@kaist.ac.kr

Insu Yun
KAIST
School of
Electrical Engineering
insuyun@kaist.ac.kr

***Abstract*—Real-Time Operating System (RTOS) is widely used in embedded systems with its various subsystems such as Bluetooth and Wi-Fi. As its functionalities grow, its attack surface also expands, exposing it to more security threats. To address this, dynamic testing techniques like fuzzing have been widely applied to embedded systems. However, for RTOS, these techniques struggle to effectively test deeply located functions within the kernel due to their complexity.**

In this paper, we present RTCon, a context-adaptive function-level fuzzer for RTOS kernels. RTCon performs function-level fuzzing on any target functions within the RTOS kernel by adaptively generating function contexts during fuzzing. Additionally, RTCon employs *Multi-layer Classification* to classify crashes by confidence levels, helping analysts focus on high-confidence crashes. We implemented the prototype of RTCon and evaluated it on four popular RTOS kernels: Zephyr, RIOT, FreeRTOS, and ThreadX. As a result, RTCon discovered 27 bugs, including 25 new bugs. We reported all of them to maintainers and received 14 CVEs. RTCon also demonstrated its effectiveness in crash classification, achieving a 92.7% precision for high-confidence crashes, compared to a 5.8% precision for low-confidence crashes.

I. Introduction

A Real-Time Operating System (RTOS) is an operating system designed for time-sensitive tasks in environments with limited resources. RTOS is optimized not only for managing time-sensitive tasks but also for supporting various peripherals critical to modern embedded applications. Thus, most RTOS projects currently provide integrated support for various features, such as Bluetooth, Wi-Fi, and Ethernet [1], [2], [3], [4].

As RTOSes expand their functionalities, their attack surface also grows, raising the risk of security vulnerabilities. More seriously, many RTOSes even lack conventional protections, such as Address Space Layout Randomization (ASLR) or stack canaries [5]. This is due to performance concerns or hardware constraints (e.g., the absence of an MMU) as RTOSes target resource-constrained embedded devices. As a result, even a simple vulnerability can lead to serious impacts, such as remote code execution or denial-of-service [6].

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241600
www.ndss-symposium.org

Despite extensive security research on embedded devices [7], [8], [9], [10], [11], [12], [13], [14], [15], two main challenges remain for RTOS kernels. First, we need a scalable method to test the various subsystems provided by the RTOS kernel. One promising approach for testing embedded devices is to emulate them and apply dynamic testing techniques, such as fuzzing. However, this approach can only cover a subset of functionalities due to limited support for various embedded boards and peripherals. We may use real-world devices to alleviate this issue, but this approach also struggles with a lack of scalability and limited resources.

Second, it is also challenging to test deeply located functions in RTOSes due to their complex execution contexts. For example, the Bluetooth subsystem, which is commonly provided by RTOSes, involves complex protocols and states. To reach deep Bluetooth functions, we need to establish connections following its complex protocol. Although many security analysts have developed manual end-to-end fuzzers to test such functions [16], [17], [18], [19], they require substantial effort for protocol analysis and implementation.

One approach to address this reachability issue is function-level fuzzing. Function-level fuzzing is an approach where, instead of fuzzing the entire system, the fuzzer directly tests specific functions (i.e., target functions) that are selected for testing. Previous works [20], [21], [22], [23] have shown that function-level fuzzing can effectively identify bugs across various libraries. However, function-level fuzzing introduces a new challenge: the absence of context. For example, in Bluetooth, most functions require a valid Bluetooth connection. Without this, these functions may crash or terminate prematurely due to missing context. As a result, function-level fuzzing may terminate early due to unintended crashes.

To address these challenges, we propose RTCon, a context-adaptive function-level fuzzer for RTOS kernels. RTCon can fuzz *any* target functions, including deeply located functions, without any need to predefine or analyze complex RTOS kernel contexts. Specifically, RTCon is not limited to fuzzing *top-layer* functions (i.e., API functions), which some previous works [20], [21] were limited to as they require analyzing function usage points. Moreover, RTCon uses *adaptive context generation* to remove the need for manually constructing contexts or going through complex analysis phases to construct

contexts. The key idea of RTCON is to generate contexts on-demand while fuzzing target functions, as inspired by X-Force [24], but *adaptively* to explore paths blocked by specific contexts. With this, RTCON can generate appropriate context values based on the operands of the branch conditions.

One big issue with function-level fuzzing is its false positives due to unknown function constraints. To mitigate this issue, RTCON uses *Multi-layer Classification* to categorize crashes based on their confidence level. The core idea is that a crash is likely to be valid if it occurs both when fuzzing the target function and its ancestor functions. If a crash occurs only in the target function, it is likely to be a false positive. Otherwise, it is likely to be an actual bug (i.e., high confidence). This classification allows security analysts to focus on high-confidence crashes. Notably, previous approaches [25], [22] rely on static analysis to extract function constraints and filter out false positives. However, these approaches are limited in extracting constraints due to obstacles that hinder static analysis, such as indirect calls and complex control flows. RTCON, on the other hand, can overcome these limitations by testing the target functions in multiple layers during fuzzing.

We implemented the prototype of RTCON and evaluated it on 4 real-world open-source RTOSes: Zephyr [26], RIOT [27], FreeRTOS [28], ThreadX [29]. As a result, RTCON found 27 bugs in total, with 25 previously unknown bugs. We reported all the bugs to the respective maintainers, and 14 of them were assigned CVE IDs. Also, RTCON can classify crashes based on their confidence levels, allowing security analysts to focus on high-confidence crashes. As a result, RTCON achieved an 92.7% precision (76 out of 82) for high-confidence crashes, compared to a 5.8% (19 out of 329) precision for low-confidence crashes.

The contributions of our work are as follows:

- We propose RTCON, a context-adaptive function-level fuzzer that fuzzes any target functions directly without any given function contexts. RTCON successfully classifies crashes based on their confidence levels, allowing security analysts to focus on high-confidence crashes.
- We evaluated RTCON on 4 real-world open-source RTOSes to demonstrate its effectiveness. Notably, RTCON could find 27 bugs, with 25 previously unknown bugs, including 14 CVE ID issued.
- We open-source RTCON to the public to encourage further research on this area: https://github.com/kaist-hacking/RTCon

II. Background

A. Real-Time Operating System

Real-Time Operating System (RTOS) is an operating system designed to handle time-critical tasks. It focuses on processing tasks within a specific time frame, rather than maximizing throughput as in general-purpose operating systems. RTOS is typically used in embedded systems since they require specific tasks to be completed before their deadlines, given the limited computational resources.

RTOS kernels provide various subsystems, including Bluetooth [1], [3], Wi-Fi [4], and network stacks [30], to support diverse devices and systems. Most RTOS projects make these features configurable [31], [32], enabling developers to build comprehensive applications suited to a wide range of systems.

However, these diverse functionalities introduce various security issues. For example, the Bluetooth and network subsystems, which are widely adopted in IoT devices, expand the attack surface through externally exposed communication channels. If we have vulnerabilities in these subsystems, attackers can exploit them to compromise the system or launch denial-of-service attacks from remote locations. Unfortunately, many RTOS kernels still lack traditional mitigation techniques (e.g., ASLR, stack canaries) due to hardware constraints or performance concerns [5]. As a result, even simple vulnerabilities can easily lead to severe implications, such as remote code execution [6].

B. Dynamic Testing on RTOS Kernels

Fuzzing [33], [34], [35] is a widely used technique that dynamically tests systems to identify vulnerabilities. Thus, several RTOS projects provide fuzzing libraries to allow security analysts to test their systems [36], [37]. Despite their effort to find bugs, it is still challenging to implement fuzzers for diverse subsystems within RTOS kernels. This is mainly because we require substantial engineering effort to manually craft fuzzers for numerous functions within complex subsystems (e.g., Bluetooth). Furthermore, analysts need to conduct comprehensive protocol analysis to test deep functions invoked at the end of complex protocol flows.

III. Motivation

A. Motivating example

In this section, we present a motivating example that demonstrates the challenges of testing deeply located functions in RTOS kernels. Figure 1 shows a code snippet of CVE-2024-8798, which is an out-of-bounds read vulnerability caused by integer underflow bugs in the Zephyr RTOS kernel. This function, `avdtp_process_configuration`, is responsible for setting local Bluetooth device configuration during an Audio/Video Distribution Transport Protocol (AVDTP) connection. It first retrieves the Streaming Endpoint object (`sep`) from the ID parsed from the Bluetooth message (`buf`) (Line 13). Then, if the `sep` is valid and the state of the `sep` is not `AVDTP_STREAMING`, it parses Initiator ID (`int_seid`) from the message (Line 24). Finally, it indirectly calls `set_configuration_ind` handler to set configuration for the remote streaming endpoint (Line 25).

This code contains integer underflow bugs at Line 13 and Line 24. This happens because the function does not check the minimum length of the message (`buf`) before pulling data from it. As a result, if an attacker sends a message shorter than 2 bytes, an integer underflow occurs in one of two vulnerable locations (Line 13 and Line 24) when attempting to pull 1 byte from an empty buffer. Then, the remaining buffer length

```
1  uint8_t net_buf_pull_u8(struct net_buf *buf) {
2      buf->len -= 1; // Integer underflow
3      return *buf->data++; // Out-of-bounds read
4  }
5
6  static void avdtp_process_configuration(struct bt_avdtp *session,
7                  struct net_buf *buf, uint8_t msg_type, uint8_t tid) {
8      if (msg_type == BT_AVDTP_CMD) {
9          int err = 0;
10         struct bt_avdtp_sep *sep;
11
12         // Get the stream endpoint from id
13         sep = avdtp_get_sep(net_buf_pull_u8(buf) >> 2);
14         if ((sep == NULL) ||
15             (session->ops->set_configuration_ind == NULL)) {
16             err = -ENOTSUP;
17         } else {
18             if (sep->state == AVDTP_STREAMING) {
19                 err = -ENOTSUP;
20             } else {
21                 uint8_t int_seid;
22                 // No check for remaining buffer size
23                 // Out-of-bounds read when it tries to pull 1 byte
24                 int_seid = net_buf_pull_u8(buf);
25                 err = session->ops->set_configuration_ind(session,
26                         sep, int_seid, buf, &error_code);
27             }
28         }
29     }
30 }
```

Fig. 1: Motivating example (CVE-2024-8798): The code shaded in red indicate vulnerable code. An integer underflow occurs when attempting to pull 1 byte from an empty buffer.

is under-wrapped, and it leads to undefined behavior in further message processing.

B. Challenges & Approaches

Unfortunately, existing approaches struggle to find deeply located bugs in RTOS kernels. In this section, we discuss technical challenges of existing approaches and our approaches to address them.

C1. Limitations of Embedded System Fuzzers

One common approach for testing embedded systems is to emulate the system or use real-world devices. Unfortunately, these methods are not typically applicable to RTOS kernels. While RTOS projects support multiple boards, emulation-based approaches [38], [39] often lack support for specific boards and peripherals. As a result, we can only test the partial features that are supported by the emulator. Obviously, it is well-known that implementing the diverse set of peripherals and boards manually is burdensome [40]. We might alleviate this burden by using on-device fuzzing [41] or semi-emulating the system [42], [43], [44], [15], but they are also struggled with scalability and performance issues.

It is also challenging to reach functions, especially those requiring complex contexts. In this example, to dynamically test the function `avdtp_process_configuration`, we need to construct the required Bluetooth connection (e.g., AVDTP connection) by following complex protocols [45]. Many developers craft specialized fuzzing harnesses to test functionality deep in the protocol stack, but it requires significant manual effort to analyze the protocol.

Our Approach: Function-level Fuzzing. To address scalability and reachability issues, RTCon uses *function-level fuzzing*. This approach enables fuzzing of target functions without requiring real-world or emulated devices. In other words, RTCon can start the fuzzing process by directly calling the target function, completely bypassing the boot-up phase, including device checks and initialization routines. Moreover, RTCon can fuzz any deeply located functions without exploring the execution paths required to reach them.

TABLE I: Comparison of function-level fuzzing tools. RTCon applies adaptive context generation to construct complex RTOS kernel contexts, reducing false positives without heavy analysis for tracking function constraints.

Tool	Target Function	Context Gen.	Constraint Scope
FuzzGen [20]	External API	-	-
FUDGE [21]	External API	-	-
FuzzSlice [23]	All	Random	0-level (Callee only)
AFGen [22]	All	Random	1-level (Caller + Callee)
RTCon	**All**	**Adaptive**	**Multi-level (n-depth)**

C2. Constructing Context for Function-level Fuzzing

Even though we can address the scalability and reachability issues using function-level fuzzing, it is still challenging to apply function-level fuzzing in large systems (e.g., RTOS kernels). This is due to the lack of context to invoke specific functions. Without this, the fuzzer cannot reach deeper code within the function and may terminate early due to unintended crashes. In this example, to reach the vulnerable code (Line 24), the fuzzer should satisfy multiple branch conditions (Lines 8, 14, and 18). Unfortunately, since these conditions are independent of the user input (`buf`), simply fuzzing the user input is insufficient to reach the vulnerable code.

As shown in Table I, some previous studies have attempted to construct contexts using various techniques (e.g., fuzzing [23], [22] or API inference [21], [20]); however, these approaches are not particularly effective for large-scale systems like RTOS kernels. For example, FuzzSlice [23] and AFGen [22] use fuzzing to construct contexts and show their effectiveness in finding bugs in libraries. However, when it comes to RTOS kernels, fuzzing is unlikely to create valid contexts, as the kernel contexts are too complicated, typically consisting of nested structures of kernel objects. On the other hand, FUDGE [21] and FuzzGen [20] attempt to construct legitimate contexts by invoking API functions. However, these approaches cannot be applied to creating internal kernel contexts, since they are typically not exposed through external APIs. For example, to fuzz the target function `avdtp_process_configuration` shown in Figure 1, we need `session` object as an argument. This object cannot be derived as it is internally managed by the kernel and not exposed through any external APIs.

Our Approach: Adaptive Context Generation. To generate contexts for any given function, RTCon employs a technique called *Adaptive Context Generation*. RTCon starts with identifying context-related variables using inter-procedural taint analysis. Then, RTCon instruments the context-generating

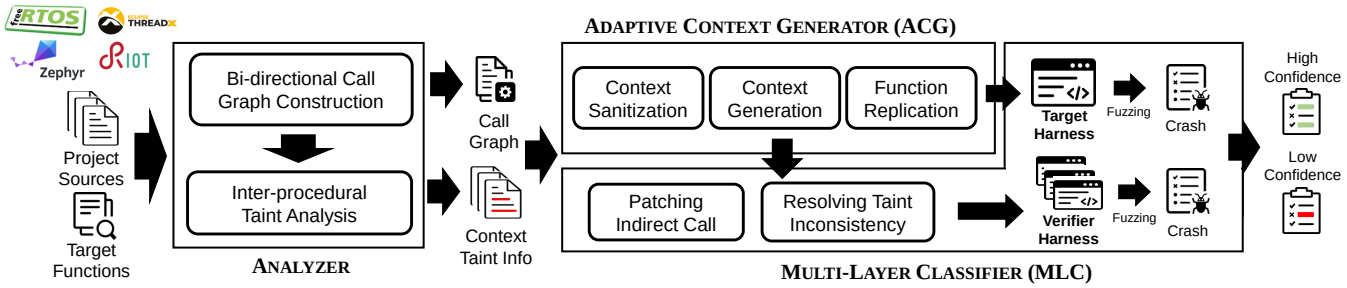


Fig. 2: Overview of RTCON

hooks at these variables. These hooks adaptively generate context at runtime, guided by branch conditions, to maximize code coverage. Notably, this method allows RTCON to eliminate the need for analyzing complex context structures.

C3. False Positives of Function-level Fuzzing

When we directly fuzz target functions, it can produce false positives due to the function constraints. These constraints are typically induced by the parent functions in the call graph. For example, even if we found a buffer overflow in a target function due to a missing length check, it might be benign if the parent function already validates the length.

Despite its importance, obtaining precise function constraints is limited. AFGen [22], a state-of-the-art function-level fuzzer, derives function constraints from control flow analysis of caller functions. Similarly, Griller [25] uses symbolic execution to extract constraints from the parent functions. While both AFGen and Griller are effective in constructing simple constraints, due to the complexity of static analysis, they struggle with complex control flows or indirect calls commonly found in RTOSes. For instance, AFGen [22] collects constraints by performing backward control flow tracing, but only to a single depth due to the complexity of nested control flows. This limitation hinders the inspection of constraints located deeper in the control flow.

Our Approach: Multi-layer Classification. Unlike previous work [22], which attempts to mitigate the constraints of identified crashes, RTCON focuses on determining whether crashes can be reproduced from parent functions for classification. To this end, RTCON uses *Multi-layer Classification* to categorize the crashes based on their confidence levels. The core idea is that if a crash is detected in a callee function, and is not also detected in its caller function, it is likely to be a false positive.

To implement this, RTCON employs multi-layer fuzzing, which performs fuzzing on both the target function and top-layer functions (i.e., other functions that have no callers and can reach the target function). RTCON regards the crashes reproduced in top-layer functions as high confidence crashes, whereas crashes detected only in the target function are considered low confidence crashes.

IV. DESIGN

A. Overview

Figure 2 depicts the overview of RTCON. RTCON consists of three components: 1) Target Analysis (ANALYZER), 2) Adaptive Context Generator (ACG), and 3) Multi-layer Classifier (MLC). Given the project source code and a list of target functions, ANALYZER constructs a call graph and uses inter-procedural taint analysis to identify context-related variables (§IV-B). Then, based on the results of ANALYZER, ACG constructs harnesses that adaptively generate contexts to explore the target functions (§IV-C). Finally, after running the harnesses, MLC classifies the detected crashes by their confidence levels (§IV-D).

B. ANALYZER

1) Call Graph Construction:

ANALYZER generates call graphs with a given list of target functions and the RTOS kernel source code. For each target function, ANALYZER constructs call graphs *bi-directionally* by recursively identifying functions that either 1) are called by (i.e., forward) or 2) call (i.e., backward) the target function with a series of function calls. ANALYZER also identifies *top-layer functions*, which are the functions in the call graph that can reach the target function but have no parent functions. These top-layer functions are later used in MLC to generate verifier harnesses.

Unlike previous works [12], [22], ANALYZER tracks indirect calls between nodes in the graph. This is necessary to handle callbacks, which are common in RTOS kernels. To achieve this, ANALYZER leverages SVF [46] to infer the possible edges between two arbitrary nodes. Internally, they resolve indirect calls through flow-insensitive pointer analysis of function pointers. While flow-insensitive pointer analysis offers high performance, it introduces false positives, resulting in a call graph that includes more functions than necessary. Since an inaccurate call graph significantly reduces the efficiency of subsequent analysis and crash classification, ANALYZER performs a simple function prototype analysis on indirect calls to optimize the call graph.

2) Inter-procedural Taint Analysis:

Intra-procedural taint analysis. ANALYZER performs taint analysis to trace the data flow of function contexts. Unlike traditional taint analysis, which taints values that are dependent

on user inputs, ANALYZER taints values that are *independent* of user inputs. This allows ANALYZER to utilize the over-approximation of taint analysis. Specifically, untainted variables are assumed to rely solely on user inputs, perfectly independent of the contexts. By doing so, when running fuzzing harnesses, RTCON can determine whether a crash is due to an uninitialized context or a potential bug. If a crash occurs due to untainted variables (i.e., user-controllable inputs), RTCON reports it as a potential bug. Otherwise, RTCON considers that it happens due to the uninitialized context and attempts to dynamically generate the missing context during fuzzing. It is important to note that RTCON leverages the generated context to explore deeper code paths, but not to detect vulnerabilities introduced by those contexts.

Figure 3 shows the intra-taint analysis results for Figure 1. In this example, the arguments `session`, `msg_type`, `tid` become the taint sources as they are not related to the user input `buf` (Lines 2-3). Taint propagates from `sep` to `state` (Line 15) as well as `session` to `set_configuration_ind` (Line 22) according to propagation rules. We detail the intra-procedural propagation rules in Algorithm 1 in Appendix.

Inter-procedural taint propagation. ANALYZER performs inter-procedural analysis using a fixed-point iteration [47]. The analysis consists of two steps: 1) ANALYZER initially over-approximates taint information of each function, and 2) it lazily updates this information based on the callee's results. First, if a callee has not yet been analyzed, ANALYZER regards its return value and reference arguments as tainted (i.e., context-related variables). This is a conservative decision as the callee's taint information is unknown. Second, once ANALYZER completes the analysis of the callee, ANALYZER revisits its callers to update taint information. This may result in removing taints from the results of the callee, which were previously assumed tainted in the first step. These changes in taints may propagate to the arguments of other function calls, which may result in reanalyzing them. This process continues until the taint information stabilizes with no further changes.

In the example of Figure 3, RTCON initially marks both the return value ① `sep` of `avdtp_get_sep` (Line 11) and ② `int_seid` of `net_buf_pull_u8` (Line 21) as tainted. Then, after analyzing `avdtp_get_sep` and `net_buf_pull_u8`, RTCON revisits the function to remove the taint from ② `int_seid` as it is not tainted in the callee function. The change in the taint status of `int_seid` propagates to an argument of `session->ops->set_configuration_ind` (Line 23), which makes it reanalyzed accordingly.

C. Adaptive Context Generator (ACG)

ACG constructs harnesses that adaptively generate contexts. To achieve this, ACG instruments hooks on the context-related variables. Specifically, ACG instruments two types of hooks for *context sanitization* and *context generation* [24], [48]. Context sanitization prevents the fuzzer from crashing due to uninitialized contexts, while context generation constructs missing contexts during fuzzing.

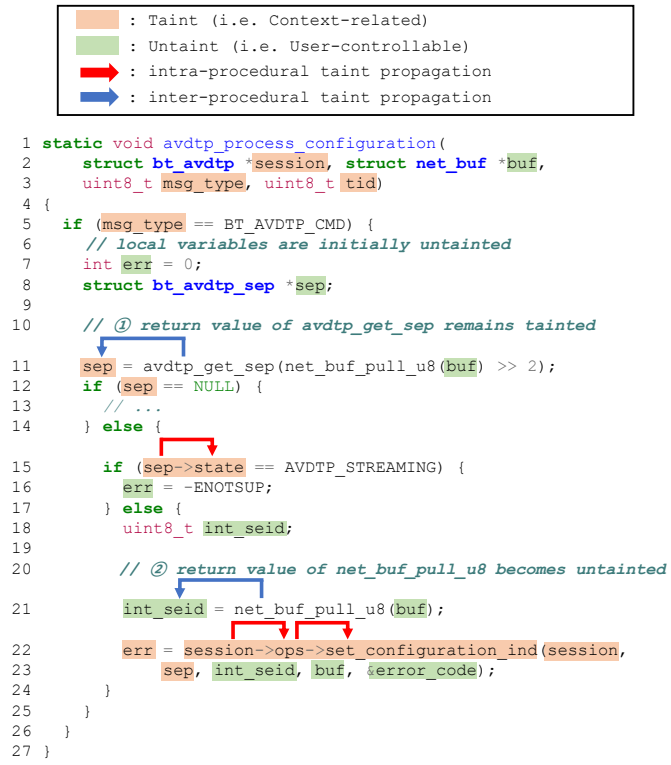
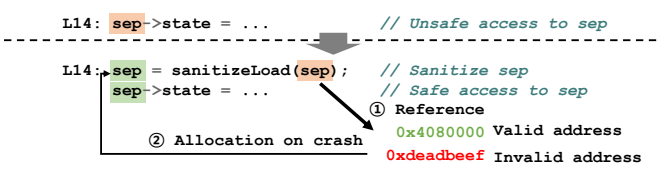


Fig. 3: Code after inter-procedural taint analysis.

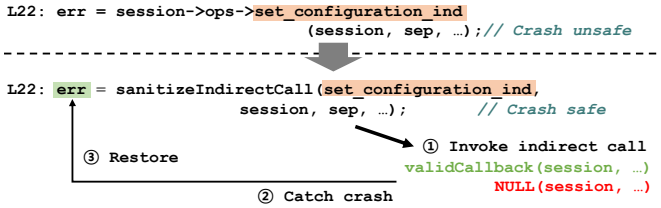
1) Context Sanitization: Context sanitization is responsible for filtering out crashes related to uninitialized contexts. From the taint analysis results from ANALYZER, ACG identifies context-related variables. If crashes occur due to these variables, ACG avoids reporting them as bugs. Instead, it avoids crashes by dynamically generating new memory regions or skipping crash points. ACG handles the following two cases in particular: load/store instructions and function calls.

Load/Store instruction. ACG sanitizes load/store instructions that are related to contexts. These instructions interrupt fuzzing when the fuzzer accesses invalid memory addresses due to uninitialized context values, such as `NULL` for pointers. To prevent harnesses from being interrupted, ACG instruments the context-related load/store instructions with sanitization hooks. These hooks check the validity of the address by trying to access the memory, and then catch any crashes that occur from invalid memory accesses. Whenever an invalid memory access is detected, the hooks dynamically allocate new memory regions and assign them to the corresponding variables to allow fuzzing to continue.

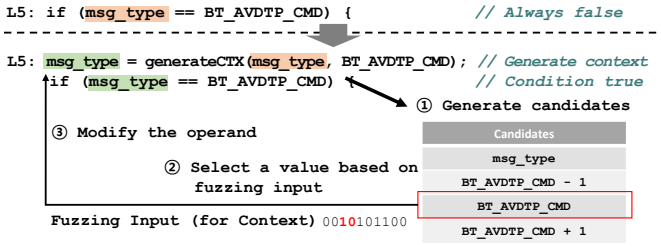
Figure 4a depicts the process of sanitizing load instructions. ACG starts with instrumenting a hook function `sanitizeLoad` to the context-related variable `sep`. Then, when the harness accesses `sep` with the load instruction, the hook first tries to access the memory. If a crash occurs from an invalid memory access, the hook catches the crash, allocates a new memory region, and assigns it to the variable. Finally, the harness continues fuzzing without crashing.



(a) An example of sanitizing a load instruction



(b) An example of sanitizing an indirect call



(c) An example of context generation

Fig. 4: Hooks for context sanitization and generation

Function Call. There are two types of function calls that ACG needs to sanitize: 1) indirect calls and 2) calls to the functions that are not instrumented. First, ACG sanitizes indirect calls with the function pointers related to contexts. These calls can interrupt fuzzing if the function pointers are uninitialized or invalid. To prevent the crashes, ACG instruments indirect calls with sanitization hooks. These hooks first check the destinations of the function pointers by trying to call them. If a destination is invalid, it catches the crash that occurred from the invalid function pointer and then skips the call to continue fuzzing.

Figure 4b depicts the process of sanitizing an indirect call. ACG instruments the hook function `sanitizeIndirectCall` to the function pointer `session->ops->set_configuration_ind`. Then, when the harness makes an indirect call, the hook first tries to call the function with the function pointer. If a crash occurs due to the invalid function pointer, the hook catches the crash, skips the call, and continues fuzzing.

Second, ACG sanitizes function calls that are not instrumented. This is because ACG can only determine if a crash is due to uninitialized context when it occurs within an instrumented function. Crashes outside instrumented functions cannot be evaluated in this way. This is because hooks, which ACG uses to detect uninitialized contexts, are only inserted with instrumentation. To prevent such crashes from interrupting fuzzing, ACG instruments the function calls with sanitization hooks. When a crash occurs, these hooks skip the function call and restore the execution state to just after the call, allowing fuzzing to continue smoothly.

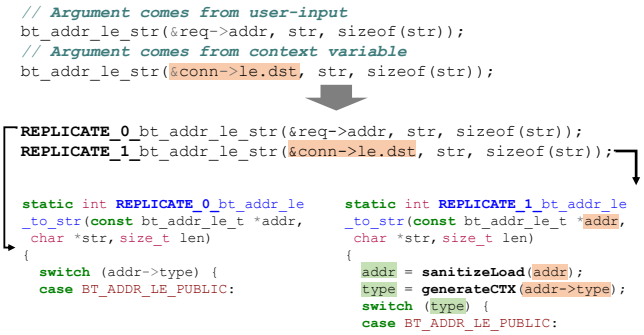


Fig. 5: An example of function replication

2) Context Generation: ACG generates contexts adaptively by capturing and using the operands involved in the context-related comparisons. This allows ACG to explore branches blocked by specific contexts. To achieve this, ACG takes three steps: 1) generating new value candidates from the operands, 2) selecting a value from the candidates, and 3) modifying the operand to the selected value.

First, ACG generates new candidate values for the comparisons. To do this, ACG starts with instrumenting a hook right before the comparison to capture the operands. Then, ACG generates new values by slightly modifying the operands, making them likely to affect the branch direction. In the example of Figure 4c, the comparison operands are `msg_type` and `BT_AVDTP_CMD`, so the candidates could be `msg_type` (i.e., no modification), `BT_AVDTP_CMD` and `BT_AVDTP_CMD`± 1.

Second, ACG uses a part of the fuzzing input as a seed to select a value from the candidates. Specifically, ACG partitions the fuzzing input into two dedicated parts: one for user input and the other for context generation. In the example, as the fuzzing input for context generation is $0b10 = 2$, ACG selects `BT_AVDTP_CMD` as the new value for the operand.

Finally, ACG modifies the operand to the selected value. ACG replaces the operand with the selected value, expecting to change the branch direction. In the example, ACG modifies the operand of `msg_type` to `BT_AVDTP_CMD` based on the fuzzing input, allowing ACG to enter the true branch.

ACG manages a mapping to ensure consistent context generation results for the same comparisons. If the comparisons have the same operands, they should have the same generated values. For example, a comparison within a loop should consistently give the same result if the comparison is independent of the loop iteration. To achieve this, ACG stores the generated values for the current location with their corresponding operands. When ACG executes the comparison, it refers to the mapping to determine whether to reuse the existing value or generate a new one.

3) Function replication: ACG can identify context-related variables more accurately as taint analysis becomes more precise. Imprecise taint analysis can lead to wrongly marking

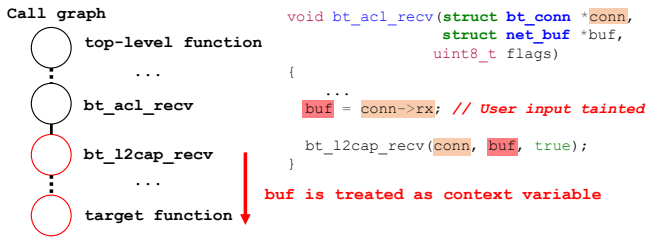


Fig. 6: An example of taint inconsistency. `buf` is marked as context-related due to taint propagation from `conn`. But, `buf` is not context-related, leading to a missed crash afterward.

variables as context-related. This can cause ACG to lose an opportunity to detect crashes due to user-controllable inputs, as ACG will wrongly try to resolve the crashes by generating contexts for user inputs.

This often occurs in function calls, where different function calls can have different sets of tainted arguments. For example, in Figure 5, the function `bt_addr_le_str` is called with two different sets of tainted arguments. The first call has the first argument untainted, while the second call has it as tainted. With the standard method of taint analysis, which over-approximates the taints, the first argument of the function will be considered tainted even if it is possible to not be. This leads to over-tainting, which decreases the precision of the analysis.

ANALYZER uses context-sensitive analysis [49] to achieve a more precise taint analysis. Specifically, ANALYZER implements 1-level context-sensitive analysis (1-CFA) through *function replication*. By replicating functions for different sets of tainted arguments, ANALYZER prepares multiple versions of the function. These versions consider all possible sets of taint sources, treating them as distinct functions when there are two or more different combinations of taint sources. The caller then selects the appropriate version based on the taint information of the arguments.

Figure 5 shows an example of function replication. In the figure, the function `bt_addr_le_str` is called with two different sets of tainted arguments, one having the first argument untainted while the other having it as tainted. ANALYZER replicates the function into two versions, `REPLICATE_0_bt_addr_le_str` and `REPLICATE_1_bt_addr_le_str`, based on the taint information of the first argument. Then, ANALYZER analyzes both versions of the function, which results in identifying different sets of tainted variables. This leads to ACG instrumenting hooks in different places for each version, allowing ACG to generate contexts precisely based on the taint information of the arguments.

D. Multi-Layer Classifier (MLC)

MLC performs multi-layer fuzzing to classify crashes detected from the target harness (i.e., the harness for the target function). The core idea is to verify these crashes by checking if they can also be triggered at higher-layer functions in the call graph, particularly at those in the top layers. To do this, MLC constructs *verifier harnesses*, which are the harnesses for the top-layer functions. Then, MLC compares crashes detected by these verifier harnesses with those found by the target harness to identify the same crashes.

Based on the results, MLC classifies the crashes into three categories: high confidence, low confidence, and unverifiable. If a verifier harness detects the same crash as the target harness does, MLC classifies it as high confidence, indicating a likely true positive. If no verifier harness detects the same crash, MLC classifies it as low confidence, suggesting a likely false positive. If the target function has no associated top-layer functions (i.e., it is the top-layer function itself), MLC classifies the crash as unverifiable.

When MLC constructs verifier harnesses, it faces two main obstacles: 1) indirect calls and 2) taint inconsistencies. First, indirect calls hinder MLC from ensuring that top-layer functions can effectively reach the target functions. This issue arises from invalid function pointers originating from uninitialized contexts, which can cause certain indirect calls to be skipped due to sanitization (see *function call* in §IV-C1). Since these calls may be essential for reaching the target functions, skipping them can prevent functions from successfully reaching the target.

Second, taint inconsistencies can cause verifier harnesses to miss crashes. These inconsistencies occur when taint analysis mistakenly over-approximates a user-controllable variable defined by the initial target function list as context-related. Specifically, taint analysis may incorrectly classify a user-controllable variable when it is stored and loaded from a context-related variable. When taint analysis incorrectly marks a user-controllable variable, the verifier harness sanitizes crashes caused by this variable, leading to missing crashes in functions further down the call graph from that variable.

Figure 6 shows an example where taint analysis incorrectly marks the user-controllable variable. `buf` is marked as a context-related variable due to the taint propagation from the context-related variable `conn`. However, `buf` is actually a user-controllable variable that is independent of the context `conn`. Since `buf` is mistakenly marked as context-related, MLC sanitizes crashes caused by `buf`, leading to a missed crash in the later execution.

To overcome the challenges, MLC uses two methods: 1) patching indirect calls and 2) resolving taint inconsistencies.

Patching indirect calls. MLC patches indirect calls to direct calls to make top-layer functions reach the target functions effectively. For each indirect call, MLC first identifies possible call targets. Then, MLC chooses functions that can lead to the target function based on the call graph. Finally, MLC patches the indirect call to direct calls to chosen functions. When there are multiple possible call targets, RTCON alternates between them in a round-robin manner.

Figure 7 shows an example of patching indirect calls. In the figure, the indirect calls described in dotted lines are patched to direct calls, as represented in red solid lines. Then, the top-layer function can effectively reach the target function through direct calls, without skipping any calls due to incorrect function pointers.

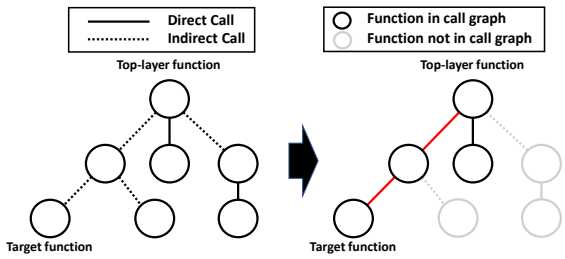


Fig. 7: An example of patching indirect calls. Indirect calls (dotted lines) are patched to direct calls (red solid lines).

Resolving taint inconsistencies. MLC resolves taint inconsistencies by additionally fuzzing at the functions that have the inconsistencies. As explained earlier, these inconsistencies occur when taint analysis mistakenly marks a user-controllable variable as context-related. To address this, MLC fuzzes at the functions that have the inconsistencies, in addition to the top-layer functions. Then, MLC uses the fuzz results from these functions to classify the crashes. Specifically, even if the top-layer functions fail to detect the same crash, MLC classifies the crash as high confidence if the functions with the inconsistencies detect it. In this way, MLC can classify the crashes more accurately by considering the crashes detected from the inconsistent functions, which may have been missed from the top-layer functions.

V. Implementation

We implemented RTCon on the Clang/LLVM framework [50] with 6.7k lines of C/C++ and Python. Specifically, we implemented Analyzer with 1.7k lines of C++ and 800 lines of Python, ACG with 3.4k lines of C/C++, and MLC with 600 lines of C/C++ and 200 lines of Python. To adapt RTCon to RTOS projects, we wrote less than 100 lines of C code per project.

We used libFuzzer [34] as the fuzz engine for our fuzzing harnesses. We used SVF [46] to construct call graphs in Analyzer. To run our target RTOSes on our x86_64 evaluation machine, we used POSIX wrappers [51], [52], [53], [54] provided by the RTOSes. In the rest of this section, we describe the implementation details of RTCon.

RTOS-specific input structures. Most RTOS functions that handle user input receive messages in RTOS-specific input structures, rather than raw bytes. These input structures are typically message buffers or network buffers that encapsulate the raw user data. For example, `net_buf`, one of the most commonly used network buffer structures in Zephyr, contains a raw user data pointer `data`, and its size `size`. To handle these input structures, we implemented simple adapter code, which is less than 25 lines of code per structure. The adapter code converts raw byte fuzzing input to these RTOS-specific input structures. Currently, RTCon supports the structures listed in Table II. We plan to support more structures in the future.

Supporting additional input structures. RTCon requires the user to manually write adapter code to support additional structures. However, the manual effort does not increase with the structure's complexity, as users only need to map the fuzzing input and its length to the corresponding fields of the input structure (e.g., Figure 13 in Appendix). All remaining fields are automatically handled as context variables.

TABLE II: List of targets and structures used in the evaluation.

RTOS	Key Structures	Kernel	Subsystem	
Zephyr	`struct net_buf`	bdca41d0	Bluetooth	bdca41d0
RIOT	`struct os_mbuf`	687a30af	Nimble	719bd3c4
FreeRTOS	`NetworkBufferDescriptor_t`	59f1c570	FreeRTOS-Plus	f940d75a
ThreadX	`NX_PACKET`	485a02fa	NetXDuo	6c8e9d1c

Making a list of target functions. To make a list of target functions, we implemented a script to automatically extract functions that take user inputs. Specifically, we extracted 430 functions that take the RTOS-specific input structures as arguments. Additionally, we manually identified 65 functions that take raw user input bytes and added them to the list. The list also includes the argument indices of the user inputs, which RTCon uses to generate harnesses.

Catching and handling crashes. ACG uses a mechanism to catch and handle crashes that occur during fuzzing. Specifically, it uses `setjmp` and `longjmp` to catch crashes from memory reference instructions and function calls. If a crash occurs, the program first jumps to the signal handler. Then, the handler restores the execution state to just before the memory reference or the function call. This allows ACG to catch the crash, take necessary actions (e.g., allocating memory), and continue fuzzing.

Blacklisting functions. ACG patches code to skip functions that are registered on a blacklist. The blacklist includes functions that disrupt fuzzing, such as `sleep` and `assert` functions. These functions can halt fuzzing by pausing execution or by causing kernel panics. To prevent these unwanted behaviors, ACG patches the functions to return immediately, allowing fuzzing to continue uninterrupted.

VI. Evaluation

In this section, we answer the following research questions:

- **RQ1.** Can RTCon find bugs in real-world RTOSes? (§VI-A)
- **RQ2.** How effective is RTCon compared to existing function-level fuzzers? (§VI-B)
- **RQ3.** How effective is RTCon compared to existing RTOS fuzzers? (§VI-C)
- **RQ4.** How effective is RTCon in classifying crashes? (§VI-D)

Experimental environment. We conducted all experiments on servers running Ubuntu 22.04, each equipped with two Intel Xeon Gold 6248R processors featuring 24 cores and 256 GB of RAM. For each fuzzing instance, we allocated one core and 4 GB of memory, running them for 24 hours five times as proposed by Klees et al. [55].

Targets. We evaluated RTCon on four widely-used RTOSes: Zephyr [26], RIOT [27], FreeRTOS [28], and ThreadX [29]

TABLE III: Discovered bugs by RTCON.

No	System	Subsystem	Status	CVE	Remote	Context	Detail
1	Zephyr	BT AP	Fixed	CVE-2024-5931	✔		No sanitization for `num_subgroups` field in `parse_recv_state`
2		BT SDP	Fixed	CVE-2024-6135	✔	✔	Mishandling of truncated packets in `sdp_client_receive`
3		BT SDP	Fixed	CVE-2024-6137	✔	✔	Missing check for the maximum number of filters in `get_att_search_list`
4		BT L2CAP	Fixed		✔	✔	Mishandling of truncated packets in `l2cap_br_info_rsp`
5		BT AVDTP	Fixed	CVE-2024-8798	✔	✔	Mishandling of truncated packets in `bt_avdtp_l2cap_recv`
6		BT ASCS	Fixed	CVE-2024-6442	✔		Missing maximum ases bounds check in `ascs_cp_rsp_add`
7		BT RFCOMM	Fixed	CVE-2024-6258	✔	✔	Mishandling of truncated packets in `rfcomm_handle_data`
8		BT OTS	Fixed	CVE-2024-6444	✔		Mishandling of truncated packets in `olcp_ind_handler`
9		BT HCI	Fixed	CVE-2024-6259	✔		Mishandling multiple advertisements in `bt_hci_le_adv_ext_report`
10		BT HCI	Reported			✔	Missing num_bis validation in `hci_le_big_complete`
11		BT Shell	Confirmed				Missing nsig and nvnd check in `bt_mesh_comp_p1_elem_pull`
12		Utils	Fixed	CVE-2024-6443			Mishandling of null starting string in `utf8_trunc`
13		LoRaWAN	Reported		✔		Missing rx_pos bounds check in `frag_transport_package_callback`
14		LoRamac-node	Reported			✔	Missing fragCounter bounds check in `FragDecoderProcess`
15	RIOT	BT HCI	Fixed		✔	✔	Missing ad_len bounds check in `_on_scan_evt`
16		BT HCI	Confirmed		✔		Mishandling of truncated packets in `_filter_uuid`
17		BT HCI	Reported		✔		Mishandling multiple advertisements in `ble_hs_hci_evt_le_ext_adv_rpt`
18		BT HCI	Reported			✔	Missing adv_handle validation in `ble_hs_hci_evt_le_adv_set_terminated`
19		BT HCI	Fixed	CVE-2024-51569			Access header before length check in `ble_hs_hci_evt_num_completed_pkts`
20		LoRaWAN	Fixed		✔	✔	Missing header length check in `gnrc_lorawan_mic_is_valid`
21		DHCP Client	Fixed	CVE-2024-52802	✔		Mishandling of truncated packets in `_parse_advertise`
22		COAP	Duplicated	CVE-2021-41040			Use vulnerable version of 3rd party library
23	FreeRTOS	DNS	Duplicated	CVE-2024-38373	✔	✔	Missing domain length validation in `DNS_ParseDNSReply`
24	ThreadX	SNMPv3	Fixed	CVE-2025-55087	✔	✔	Mishandling of truncated packets in `_nx_snmp_version_3_process`
25		HTTP	Fixed	CVE-2025-55085	✔	✔	Mishandling of truncated packets in `_nx_web_http_client_process_header_fields`
26		DHCPv6	Fixed	CVE-2025-55086	✔	✔	Mishandling of truncated packets in `_nx_dhcpv6_process_server_duid`
27		DHCPv6	Confirmed		✔	✔	Missing malicious label length check in `_nx_dhcpv6_name_string_unencode`

along with their respective subsystems. However, due to the large size of RTOSes, we could not evaluate RTCON on all subsystems. Instead, we evaluated RTCON on one of the most representative subsystems for each RTOS: Bluetooth and network stacks. Table II shows the specific versions of the RTOS kernels and subsystems used in our experiments. Notably, we also tested Nimble [3] and NetXDuo [30], which are the external libraries for RIOT and ThreadX, respectively. These libraries implement the Bluetooth and network stacks for RIOT and ThreadX. For FreeRTOS, we tested the network stack libraries implemented in FreeRTOS-Plus [4]. Specifically, we tested the TCP/UDP protocols along with their upper layers.

While we could not evaluate RTCON on all subsystems, we strongly believe that RTCON can be extended to other subsystems with minimal effort. Specifically, we can extend RTCON to other subsystems by simply adding target functions to the list of functions to fuzz, and specifying the input format of the target functions. Furthermore, we can even extend RTCON to other targets, such as general libraries, as we show the details in §VII-C.

Analysis time. We detail the average analysis time of ANALYZER and ACG for each project in Table X in the Appendix. The analysis time generally scales with the size of the target RTOS codebase and the complexity of the target function. On average, it takes about 10 minutes to analyze the kernel call graph, with an additional 140% time required for taint analysis and ACG. Notably, both ANALYZER and ACG need to be executed only once per harness.

A. Effectiveness of Finding New Bugs

To evaluate the effectiveness of RTCON in finding bugs in real-world RTOSes, we conducted a 24-hour fuzzing campaign on our target RTOSes. As a result, RTCON found a total of 27 bugs across 4 RTOSes, with 25 previously unknown bugs. Table III provides detailed information on the bugs found by RTCON. Specifically, RTCON found 14 bugs in Zephyr, 8 in RIOT, 1 in FreeRTOS, and 4 in ThreadX. We reported all the bugs to the respective maintainers. They confirmed and fixed 20 of them with 14 CVEs assigned.

Among the bugs found by RTCON, we manually confirmed that 20 bugs are reachable by remote attackers. In other words, attackers can send malicious packets to trigger these vulnerabilities, leading to potential remote code execution. This result demonstrates the effectiveness of RTCON in finding new bugs in real-world RTOSes and their security impacts.

RTCON could achieve these successful results thanks to its support for adaptive context generation. As shown in Table III, we found that 15 bugs required specific contexts to reach the vulnerable code. For example, CVE-2024-6135 in Zephyr, found in the Bluetooth Service Discovery Protocol (SDP), requires valid contexts for both a Logical Link Control and Adaptation Protocol (L2CAP) channel and an SDP channel to trigger. RTCON can find this bug by generating the necessary contexts without manually constructing these channels. We present additional detailed case studies in §A of Appendix to demonstrate the effectiveness of RTCON in finding bugs in RTOS kernels with specific contexts.

B. Comparison with Function-Level Fuzzers

In this section, we compare the effectiveness of RTCON in testing RTOS kernels with function-level fuzzers. In particular, we compare RTCON with FuzzSlice [23], a state-of-the-art function-level harness generator, and FuzzGen [20], an API harness generator originally targeting AOSP libraries. We evaluated them to demonstrate the effectiveness of RTCON in comparison to existing function-level fuzzers, even though they are not designed for testing RTOSes. There are also other tools such as FUDGE [21] and AFGen [22], but we could not

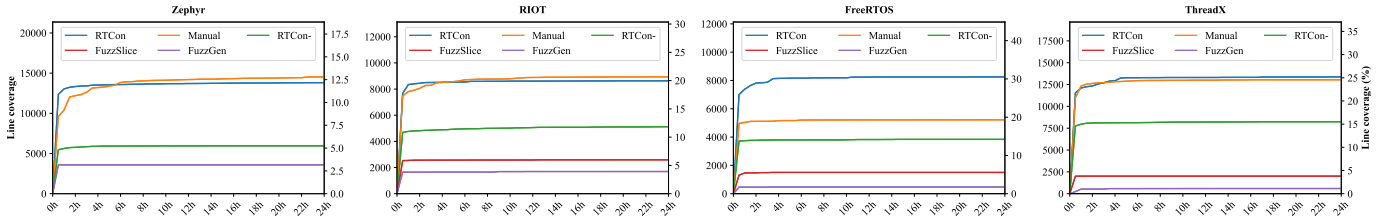


Fig. 8: Line coverage measured in four RTOSes: Zephyr, RIOT, FreeRTOS, and ThreadX.

evaluate them since they are not publicly available. Instead, we also compared RTCON with manually crafted harnesses for the target functions.

Unlike RTCON and FuzzSlice, FuzzGen requires *consumers*, which are the applications that use the target function. FuzzGen analyzes these applications to find the usages of the target function and generate harnesses. In this evaluation, we provided sample and test code from each RTOS project as consumers for FuzzGen.

Figure 8 depicts the graphs of line coverage achieved by RTCON, our manual harnesses, RTCON$^-$ (for ablation study), FuzzSlice, and FuzzGen in four RTOS kernels. More detailed results are provided in Table VIII of Appendix. The table includes the average line coverage, as well as Mann-Whitney U test results [55] to determine the statistical significance between RTCON and the other fuzzers.

Comparison with FuzzSlice and FuzzGen. Figure 8 shows the line coverage that RTCON and the harnesses generated by FuzzSlice and FuzzGen achieved. To summarize, RTCON achieved significantly higher coverage than both FuzzSlice and FuzzGen across all RTOS kernels. This is because the generated harnesses struggled with unintended crashes, primarily caused by uninitialized kernel subsystems and incomplete inference of data structures. Furthermore, as function arguments become more complex, the fuzzing input space grows exponentially, making it unlikely to create interesting inputs. In the case of FuzzGen, while it successfully caught the functions used in subsystem initialization, there were limitations in identifying callback functions that are not explicitly invoked. However, RTCON could bypass the inference of complex structures and API specifications through its adaptive context generation. This allowed RTCON to achieve high coverage while avoiding early termination of the harnesses.

Comparison with manual harnesses. To evaluate the effectiveness of RTCON in generating contexts, we compared RTCON against our manual harnesses. We built these manual harnesses by analyzing the API usage of the kernel and subsystems, mostly referring to sample and test code. As a result, we built harnesses for each RTOS, initializing the kernel and creating temporary contexts, such as Bluetooth connection, before testing the target functions. In particular, we used the Bumble [56] Bluetooth emulator to establish a mockup Bluetooth connection.

Figure 8 shows the coverage results of RTCON and manual harnesses. The results show that RTCON achieved similar or higher coverage than manual harnesses for FreeRTOS and ThreadX. However, in the case of Zephyr and RIOT, RTCON recorded about 1%p lower coverage. This small difference comes from manual harnesses gaining coverage by executing kernel-specific functions like scheduling and multi-threading, which are skipped by RTCON. While both RTCON and manual harnesses can achieve high coverage in function-level fuzzing, RTCON removes the need for the manual effort required to construct context, enhancing scalability.

Ablation study. To evaluate the effectiveness of adaptive context generation, we conducted an ablation study. For this, we implemented RTCON$^-$, which disables the adaptive context generation. Figure 8 shows the coverage results of RTCON and RTCON$^-$. The results show that RTCON achieved 7%p, 8%p, 16%p, and 5%p higher coverage than RTCON$^-$ for Zephyr, RIOT, FreeRTOS, and ThreadX, respectively. This indicates that the generated contexts are crucial in exploring the target functions. Notably, RTCON outperforms RTCON$^-$ more significantly in FreeRTOS. This is because a considerable number of tested functions in FreeRTOS validate input arguments at the beginning of the function using `assert`, preventing RTCON$^-$ from exploring the functions further. However, RTCON can bypass them by adaptively setting up necessary conditions.

Bug findings. We evaluated our manual harnesses, RTCON$^-$, FuzzSlice, and FuzzGen to see if they can detect new bugs discovered by RTCON. Table IV shows the result. Out of the 27 bugs, our manual harnesses and RTCON$^-$ could detect 20 and 13 bugs, respectively. In contrast, FuzzSlice and FuzzGen either failed to generate harnesses or did not detect any bugs. This is because most of the harnesses generated by FuzzSlice and FuzzGen terminated prematurely due to unintended crashes from the incorrect inference of data structures. Specifically, the incomplete analysis of complex structures led to incorrect values being populated within both context and input structures. As a result, most harnesses crashed upon attempting to reference invalid pointers within these structures. However, given that FuzzSlice and FuzzGen are not designed for testing RTOS kernels, this does not imply they are generally ineffective.

RTCON$^-$ was able to find relatively shallow bugs, but could not find deeper bugs. For example, in the case of CVE-2024-6444 (Bug #8), RTCON$^-$ was able to find this bug because there were no references to context variables before reaching the vulnerable code. However, it could not find most bugs that

TABLE IV: Bug finding capability of RTCON compared to other function-level fuzzers. The table shows that FuzzSlice and FuzzGen could not detect any bugs. However, note that these fuzzers are not designed for testing RTOSes.

System	Zephyr														RIOT								FreeRTOS	ThreadX			
Bug ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
RTCON	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Manual	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	-	-	-	-	✔	✔	✔	✔	✔	✔	✔	-	✗	✗	✔	✔	✔
RTCON⁻	✗	✗	✗	✗	✗	✔	✗	✔	✔	✔	✗	✔	✔	✗	✔	✔	✔	✗	✗	✔	✗	✗	✗	✗	✔	✔	✔
FuzzSlice	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
FuzzGen	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

-: The harness could not be generated.

TABLE V: Comparison of RTCON with RTOS fuzzers.

RTOS	RTCON (E/C/B)	HOEDUR (E/C/B)	SFuzz (E/C/B)
Zephyr	7,336 / 288 / 10	3,144 / 99 / 1	256 / 64 / 4
RIOT	2,536 / 69 / 6	803 / 87 / 1	216 / 20 / 0
FreeRTOS	2,403 / 37 / 1	947 / 84 / 0	30 / 8 / 0
ThreadX	3,972 / 101 / 3	680 / 105 / 0	130 / 20 / 0
Total	**16,247 / 495 / 20**	5,574 / 375 / 2	632 / 112 / 4

E: # of edges covered. / **C**: # of unique crashes. / **B**: # of identified bugs.

TABLE VI: Multi-layer classification results of detected crashes. Specifically, 15 bugs were identified as high-confidence through taint inconsistency resolution.

	V	Total	High	Low	Unverifiable
Zephyr	3.6	**64** / 258 (24.8%)	**52** (**15**) / 57 (91.2%)	**6** / 195 (3.1%)	**6** / 6 (100.0%)
RIOT	1.2	**33** / 39 (84.6%)	**20** / 20 (100.0%)	**0** / 4 (0%)	**13** / 15 (86.7%)
FreeRTOS	1.1	**11** / 35 (31.4%)	**0** / 1 (0%)	**11** / 30 (36.7%)	**0** / 4 (0%)
ThreadX	1.6	**7** / 111 (6.3%)	**4** / 4 (100.0%)	**2** / 100 (2.0%)	**1** / 7 (14.3%)
Total	1.9	**115** / 443 (26.0%)	**76** (**15**) / 82 (92.7%)	**19** / 329 (5.8%)	**20** / 32 (62.5%)

V: Average number of verifier harnesses created per target function.

require specific contexts.

Manual harnesses were able to find a substantial number of bugs; however, they require significant manual effort to construct the necessary contexts. For some subsystems (e.g., Bluetooth Mesh and LoRaWAN), we could not configure the harnesses, so they were excluded from the comparison.

C. Comparison to state-of-the-art RTOS fuzzers

Experimental setup. To see how effective RTCON is in testing RTOS kernels compared to existing RTOS fuzzers, we evaluate RTCON with SFuzz [12], a state-of-the-art sliced-based RTOS fuzzer, and HOEDUR [14], a multi-stream input-based firmware fuzzer. As SFuzz only supports the binaries compiled for ARM and MIPS architectures, we compiled the RTOS kernels for ARM architecture with the same configurations as in previous experiments. Additionally, we manually included source and sink functions for each RTOS, which SFuzz uses to find vulnerable paths in the target binaries.

Result. Table V shows a comparison result of the number of covered edges, unique crashes found, and identified bugs. RTCON explores significantly more edges and discovers more crashes than both HOEDUR and SFuzz. HOEDUR, which basically adopts a firmware fuzzing approach, provides inputs to the emulated RTOS kernel via hardware registers (e.g., MMIO registers) to fuzz the interrupt handler. However, similar to API fuzzers, it has limitations in exploring the functions that require specific contexts. For instance, while HOEDUR may successfully invoke the Bluetooth HCI command handler, it cannot reach upper-layer handlers (e.g., Bluetooth SMP) due to the absence of necessary connection information.

Compared to RTCON and HOEDUR, SFuzz achieves lower coverage, primarily due to two main reasons. First, SFuzz slices code snippets and prunes the unnecessary paths by patching the code. They skip calls and modify branches to fixed jumps, which results in fewer edges being explored. Second, SFuzz limits fuzz testing to potentially vulnerable paths between source and sink functions, meaning it cannot explore edges outside these paths. For example, in the case of FreeRTOS, SFuzz only found two potential vulnerable paths, since FreeRTOS performs direct memory operation rather than using the vulnerable sink functions (e.g., `memcpy`).

It is worth noting that SFuzz and HOEDUR are designed for different purposes; SFuzz primarily targets user applications running on RTOSes (e.g., COTS router, printer firmware), while RTCON targets RTOS kernels themselves. HOEDUR, on the other hand, focuses on fuzzing through external peripheral inputs, which contrasts with our function-level fuzzing approach. Due to these differences, we cannot claim that RTCON can replace SFuzz or HOEDUR. However, this evaluation demonstrates that an approach like RTCON— applying function-level fuzzing to RTOS — has unique advantages in testing deep RTOS kernel code compared to existing RTOS fuzzers.

D. Effectiveness of Multi-layer Classification

In this section, we evaluate the effectiveness of RTCON in classifying the crashes. For each target function, we ran both the verifier and target harnesses for a total of 24 hours.

Result. Table VI shows the result of the multi-layer classification. The table shows the following: 1) the average number of verifier harnesses created per target function, 2) the total number of unique crashes detected by a target harness, and 3) the number of crashes at each confidence level, as classified by RTCON. Bold numbers represent unique valid crashes, which are manually confirmed.

As a result, RTCON constructed an average of 1.9 verifier harnesses for each target function. With these verifier harnesses, RTCON detected a total of 443 unique crashes with 115 valid crashes. Specifically, RTCON detected 258, 39, 35, and 111 unique crashes at target functions from Zephyr, RIOT, FreeRTOS, and ThreadX, respectively. Among them, we found

that 64, 33, 11, and 7 crashes are valid. Through the multi-layer classification, RTCON classified 52, 20 and 4 crashes as high-confidence bugs, achieving precision of 91.2%, 100.0% and 100.0% in Zephyr, RIOT and ThreadX. However, RTCON could not find high-confidence bugs in FreeRTOS. A total of 32 crashes could not be verified due to the absence of verifier harnesses, of which 20 were valid crashes.

Specifically, 15 bugs were identified as high-confidence through taint inconsistency resolution. This typically occurs when user input is stored in a context variable and later copied from it (as described in Figure 6). Since RTCON detects such mis-taints and reports them from the highest possible layer of the call graph, it can effectively validate these bugs.

In the following, we present false positives and underclassified bugs. For more comprehensive analysis, we also further evaluate RTCON on previously known bugs (see §C).

False positives. Even when crashes are reproduced from top-layer functions, RTCON may report false positives for two reasons. First, false positives can occur in functions that receive user inputs assumed to be trusted. For example, among the five false positives detected in Zephyr, three were in a function that handles messages from the local Bluetooth host. Although we reported these crashes to the maintainers, they did not consider them bugs, as the inputs are assumed to be trusted and well-formed. Second, incomplete taint propagation caused by double-pointer variables can also lead to false positives. Specifically, RTCON currently does not handle taint propagation through indirect memory access using double pointers. This can make RTCON miss sanitizing invalid crashes and result in false positives. In our result, two other false positives in Zephyr are caused by this reason.

Underclassified bugs. While RTCON could classify a fair amount of actual bugs as high confidence, it still missed some bugs. This is because verifier harnesses could not reach certain functions due to road-blocking routines, such as checksum validation. §B provides detailed examples of such routines observed in Zephyr and FreeRTOS. In our result, all 11 crashes in FreeRTOS were located in DNS packet processing functions. To verify these bugs, RTCON attempted to generate valid messages starting from the top-layer Ethernet frame handler. However, due to checksum validation in each protocol layer, RTCON could not generate inputs that could pass the checksum within the limited time frame. Instead, RTCON reported the crashes were reproduced from middle-layer handlers, like UDP packet processing functions.

This issue paradoxically highlights the need for direct fuzzing at the low-layer function. As shown in the results, fuzzing from the top-layer function (e.g., API and main) does not ensure reaching the functions that are deeply located in the program. Thus, this would lead the fuzzer to miss potential bugs in these functions, resulting in false negatives. In contrast, RTCON addresses this limitation by directly fuzzing the low-layer functions, effectively detecting bugs that other fuzzers may miss.

VII. Discussion

A. Imprecise Pointer Analysis

When fuzzing at top-layer functions, RTCON forcibly traces indirect calls in the call graph. However, this can result in unwanted crashes. As mentioned in §IV-B, this is mainly due to the flow-insensitive nature of our analysis, which RTCON uses to trace indirect calls at corresponding call sites. While some previous works [57], [58] proposed flow-sensitive pointer analysis to improve precision, RTCON currently adopts flow-insensitive pointer analysis due to the high complexity of such analysis.

While this simplifies our process, it may reduce the fuzzing efficiency of the verifier harnesses. To address this, RTCON offers additional hooking points that allow a user to validate the legitimacy of an indirect call at call sites. Once a hook function is registered, RTCON executes it before invoking an indirect call to perform user-defined validity checks.

We note that this approach is unlikely to cause false positives. Crashes caused by imprecise pointer analysis typically originate from context variables or occur at random code locations. However, RTCON filters out such cases by sanitizing context-related crashes and comparing the call stacks between the verifier and target harnesses.

B. False Positives & False Negatives

Unlike classical fuzzing, RTCON can detect more vulnerabilities as it needs fewer constraints to satisfy (see §VI-A). This is because RTCON can begin fuzzing directly from a function without any context. However, as RTCON is fundamentally based on fuzzing, it inherits the limitations of fuzzing. Moreover, to enable function-level fuzzing, RTCON also introduces several trade-offs that result in both false positives and false negatives.

Infeasible Contexts & Inputs. As RTCON constructs contexts and inputs arbitrarily without checking their feasibility, it may report false positives. For example, RTCON may infer that two context variables differ even when they are, in fact, aliases. Although RTCON does not directly report crashes caused by such invalid context variables, it may still report false crashes along paths that appear reachable only because of these infeasible contexts. This also happens to an input if it is incorrectly assumed to be fully controllable (see §VI-D).

Uninitialized Context Variables. RTCON also cannot detect bugs caused by uninitialized context variables. This is because RTCON assumes that all context variables are properly initialized. This assumption enables RTCON to perform function-level fuzzing without any contexts, but it can also suppress genuine failures, leading to false negatives.

Over-tainting. Over-tainting from the context variables can mask real bugs (see §IV-B2), leading to false negatives. RTCON adopts this conservative approach to suppress crashes caused by context variables. As a result, bugs that depend on both input values and context variables (e.g., arithmetic operations involving both) may be missed.

Uninstrumented Code. Because RTCON detects only crashes within instrumented code, it may miss bugs in uninstrumented code (e.g., linked libraries). To detect these bugs, we need to include target libraries in its analysis.

Limited Implementation. We also observed that RTCON may cause false positives due to its limited implementation. As discussed in §VI-D, the current prototype of RTCON does not yet handle taint propagation for double-pointer variables. In such cases, RTCON misses to sanitize crashes caused by invalid context variables, which can lead to false positives.

C. Effectiveness of Finding Bugs in General Libraries

Experimental setup. While RTCON primarily targets RTOS kernels, it can be easily extended to general-purpose libraries. To see the applicability of RTCON to general libraries, we evaluated whether RTCON could detect previously known vulnerabilities in seven general libraries. The target libraries and vulnerabilities were selected based on those discovered by AFGen, the state-of-the-art function-level fuzzer. Since the source code of AFGen is not publicly available, we could not fully reproduce the results of AFGen. However, as AFGen provided the generated harnesses for the discovered vulnerabilities, we could run the harnesses to reproduce the results. Also, we could infer the vulnerable versions of the libraries using the publicly disclosed CVE information and the harness code.

Result. RTCON was able to detect 24 out of 26 bugs across seven libraries (details in Table IX of Appendix). In particular, CVE-2022-30858 of `ngiflib`, a buffer overflow vulnerability when processing the file contents, could be detected by RTCON by adding a simple hook of fewer than five lines of code. Similarly to what AFGen does, this hook redirects the fuzzing input as file content to the target function.

Of the two undetected bugs, CVE-2022-34526 in `libtiff` was not reproduced by either AFGen or RTCON. The other, CVE-2023-23054 in `libming`, RTCON could not detect it because it is an infinite loop vulnerability that RTCON is not designed to detect. Before reaching the timeout required to detect the bug, RTCON reaches the maximum limit on the number of times context generation hooks can be invoked, which is designed to prevent performance degradation caused by infinite loops resulting from generated contexts.

D. Challenges on Function Level Fuzzing RTOS Kernels

Applying function-level fuzzing to RTOS kernels poses its own unique challenges compared to fuzzing general-purpose libraries. We demonstrate their challenges by evaluating two existing tools, FuzzGen [20] for API fuzzing and Angr [59] for under-constrained symbolic execution. Our evaluation shows two challenges in applying them to RTOS kernels.

Lack of Exposed APIs. RTOS kernels lack exposed APIs to construct necessary contexts for fuzzing target functions. In practice, most harnesses produced by FuzzGen were limited to externally visible kernel initialization calls (e.g., `bt_enable`). Therefore, they could not construct specialized contexts required by the targets (e.g., `LoRaWAN` stacks or `RFCOMM` sessions in Figure 9 and Figure 10). This contrasts with general-purpose libraries, which commonly provide explicit context-building APIs (e.g., `EVP_EncryptInit` in OpenSSL [60]).

Difficulty in Constructing Complex Contexts. Constructing complex contexts accurately is fundamentally challenging for both function-level fuzzers and symbolic executors. RTOS kernels contain deeply nested data structures and rely heavily on indirect calls, which both of them leave many of the constraints needed to reach target functions unresolved. According to our analysis, symbolic execution failed to propagate constraints along the call paths into the target, and indirect calls (e.g., `dev->driver->set` in Figure 9) remained unresolved, causing early termination of analysis.

VIII. Related work

Real-Time OS Fuzzing. Despite extensive research [7], [8], [9], [10], [11], [41], [61], [62] focused on security analysis of embedded devices, a few [12], [63], [13], [14] works specifically targeted RTOS kernel and the applications running on them. Li et al. [63] abstracts the HAL functions to rehost MCU firmware based on RTOS, while HOEDUR [14] proposes firmware-aware fuzzing with a multi-stream input representation. SFuzz [12] slices code snippets from the call graph and emulates small portions of the binary to perform micro-fuzzing. Rtkaller [13] applies task-based fuzzing to rt-Linux, but it is closely related to Linux kernel fuzzing. While most of these works show their effectiveness in identifying bugs, they require engineering effort to rehost the firmware or emulate the part of the system. In contrast, RTCON tests RTOS kernels without actual devices or emulation with function-level fuzzing and adaptive context generation.

Automatic Harness Generation. Instead of dynamically generating contexts, previous works [20], [21], [64], [65], [23] have focused on statically generating contexts to reach deep internal states. FuzzGen [20] builds fuzz drivers for the target libraries by inferring their interfaces and analyzing API dependency graphs. Similarly, FUDGE [21] builds fuzz drivers by analyzing the usage patterns of the target libraries. GraphFuzz [64] models sequences of executed functions as a dataflow graph to test low-level library APIs, while APICraft [65] builds fuzz drivers by collecting control and data dependencies for API functions and combining these dependencies. These approaches leverage existing codebases to build API fuzz drivers. While they can construct legitimate contexts based on APIs, they struggle to generate contexts for fuzzing arbitrary functions.

Recent works [23], [22], [25] build fuzz drivers using static analysis. Specifically, FuzzSlice [23] builds fuzz drivers by creating small, compilable code slices at the function level. AFGen [22] takes a bottom-up approach, building fuzz drivers that target internal functions while collecting constraints from call sites. Similarly, Griller [25] performs symbolic execution to extract constraints from the parent functions. These approaches have demonstrated effectiveness in finding bugs. However, unlike RTCON, they heavily rely on static analysis

to reduce false positives and struggle to populate context structures for complex kernel objects.

IX. Conclusion

In this paper, we presented RTCon, a context-adaptive function-level fuzzer for RTOS kernels. RTCon performs function-level fuzzing on any target functions within the RTOS kernel without any given function contexts. To achieve this, RTCon adaptively generates the necessary contexts on demand while fuzzing target functions. Additionally, RTCon uses *Multi-layer Classification* to classify crashes based on confidence, allowing security analysts to focus on high-confidence crashes. We implemented the prototype of RTCon and evaluated it on four popular RTOS kernels: Zephyr, RIOT, FreeRTOS, and ThreadX. As a result, RTCon discovered 27 bugs, with 25 previously unknown bugs, including 14 CVE ID issued. RTCon also demonstrated its effectiveness in crash classification, achieving an 92.7% precision for high-confidence crashes, compared to a 5.8% precision for low-confidence crashes.

Acknowledgment

We thank the anonymous reviewers and shepherd for providing valuable feedback. This work was supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2024-00331961).

References

[1] Z. Project, "Zephyr project documentation - bluetooth," https://docs.zephyrproject.org/latest/connectivity/bluetooth/index.html, 2025, accessed: 2025-12-15.

[2] ——, "Zephyr project documentation - ethernet," https://docs.zephyrproject.org/latest/connectivity/networking/api/ethernet.html, 2025, accessed: 2025-12-15.

[3] A. Mynewt, "Ble user guide - apache mynewt latest documentation," https://mynewt.apache.org/latest/network/, 2025, accessed: 2025-12-15.

[4] FreeRTOS™, "Freertos plus libraries," https://www.freertos.org/Documentation/03-Libraries/02-FreeRTOS-plus/01-Introduction, 2025, accessed: 2025-12-15.

[5] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," in *Proceedings of the 4th European Symposium on Security and Privacy (EuroS&P)*, Stockholm, Sweden, Jun. 2019.

[6] B. Seri, G. Vishnepolsky, and D. Zusman, "Critical vulnerabilities to remotely compromise vxworks, the most popular rtos," *White Paper, ARMIS, URGENT/11*, 2019.

[7] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[8] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2020.

[9] I. Angelakopoulos, G. Stringhini, and M. Egele, "Firmsolo: Enabling dynamic analysis of binary linux-based iot kernel modules," in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[10] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl:high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[11] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "Halucinator: Firmware re-hosting through abstraction layer emulation," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[12] L. Chen, Q. Cai, Z. Ma, Y. Wang, H. Hu, M. Shen, Y. Liu, S. Guo, H. Duan, K. Jiang, and Z. Xue, "Sfuzz: Slice-based fuzzing for real-time operating systems," in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, Nov. 2022.

[13] Y. Shen, H. Sun, Y. Jiang, H. Shi, Y. Yang, and W. Chang, "Rtkaller: State-aware task generation for rtos fuzzing," vol. 20, no. 5, Apr. 2021.

[14] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel, and T. Holz, "Hoedur: Embedded firmware fuzzing using multi-stream inputs." in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[15] A. Mera, C. Liu, R. Sun, E. Kirda, and L. Lu, "Shift: Semi-hosted fuzz testing for embedded applications," in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

[16] M. E. Garbelini, V. Bedi, S. Chattopadhyay, S. Sun, and E. Kurniawan, "Braktooth: Causing havoc on bluetooth link manager via directed fuzzing," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[17] D. Heinze, J. Classen, and M. Hollick, "Toothpicker: Apple picking in the ios bluetooth stack," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, Boston, MA, Aug. 2020.

[18] H. Park, C. K. Nkuba, S. Woo, and H. Lee, "L2fuzz: Discovering bluetooth l2cap vulnerabilities using stateful fuzz testing," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Jun. 2019.

[19] J. Jang, M. Kang, and D. Song, "Reusb: Replay-guided usb driver fuzzing." in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[20] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[21] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 24th European Software Engineering Conference (ESEC) / 27th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Tallinn, Estonia, Aug. 2019.

[22] Y. Liu, Y. Wang, X. Jia, Z. Zhang, and P. Su, "Afgen: Whole-function fuzzing for applications and libraries," in *Proceedings of the 45th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2024.

[23] A. Murali, N. Mathews, M. Alfadel, M. Nagappan, and M. Xu, "Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing," in *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, Lisbon, Portugal, Apr. 2024.

[24] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-Force:Force-Executing binary programs for security applications," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[25] S. Muralee, J. Srinivasan, A. Pillai, A. Bianchi, A. Machiry, G. Vigna, and C. Kruegel, "Griller - a framework for under constrained fuzzing," 2022, accessed: 2025-12-15.

[26] Z. Project, "Zephyr project," https://www.zephyrproject.org, 2025, accessed: 2025-12-15.

[27] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "Riot os: Towards an os for the internet of things," in *2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*. IEEE, 2013, pp. 79–80.

[28] FreeRTOS™, "Freertos™ - real-time operating system for microcontrollers and small microprocessors," https://www.freertos.org/, 2025, accessed: 2025-12-15.

[29] T. E. Foundation, "Azure rtos is now eclipse threadx," https://threadx.io/, 2025, accessed: 2025-12-15.

[30] E. ThreadX, "Eclipse threadx netxduo," https://github.com/eclipse-threadx/netxduo, 2025, accessed: 2025-12-15.

[31] Z. Project, "Zephyr project documentation - configuration system," https://docs.zephyrproject.org/latest/build/kconfig/index.html, 2025, accessed: 2025-12-15.

[32] R. OS, "Kconfig in riot — riot documentation," https://doc.riot-os.org/kconfig-in-riot.html, 2025, accessed: 2025-12-15.

[33] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.
[34] L. Project, "libfuzzer - a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, 2025, accessed: 2025-12-15.
[35] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, Boston, MA, Aug. 2020.
[36] Z. Project, "Zephyr project documentation - fuzzing," https://docs.zephyrproject.org/latest/samples/subsys/debug/fuzz/README.html, 2025, accessed: 2025-12-15.
[37] R. OS, "Fuzzing utilities — riot documentation," https://doc.riot-os.org/group__sys__fuzzing.html, 2025, accessed: 2025-12-15.
[38] F. Bellard, "Qemu, a fast and portable dynamic translator." in *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, Jun. 2005.
[39] N. A. Quynh and D. H. Vu, "Unicorn: Next generation cpu emulator framework," Aug. 2015.
[40] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti, and W. Robertson, "Sok: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 16th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Hong Kong, China, Jun. 2016.
[41] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.
[42] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.
[43] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated rehosting," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Beijing, China, Sep. 2019.
[44] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Proceedings of the NDSS Workshop on Binary Analysis Research (BAR)*, San Diego, CA, Feb. 2018.
[45] I. Bluetooth SIG, "Part e. host controller interface functional specification," https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core-54/out/en/host-controller-interface/host-controller-interface-functional-specification.html, 2025, accessed: 2025-12-15.
[46] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.
[47] R. P. Agarwal, M. Meehan, and D. O'regan, *Fixed point theory and applications*. Cambridge university press, 2001, vol. 141.
[48] D. Maier and L. Seidel, "Jmpscare: Introspection for binary-only fuzzing," in *Proceedings of the NDSS Workshop on Binary Analysis Research (BAR)*, Virtual, Feb. 2021.
[49] M. Might, Y. Smaragdakis, and D. Van Horn, "Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 305–315.
[50] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.
[51] Z. Project, "Zephyr project documentation - native simulator - native_sim," https://docs.zephyrproject.org/latest/boards/native/native_sim/doc/index.html, 2025, accessed: 2025-12-15.
[52] R. OS, "Posix wrapper for riot — riot documentation," https://doc.riot-os.org/group__posix.html, 2025, accessed: 2025-12-15.
[53] FreeRTOS™, "Posix/linux simulator demo for freertos using gcc," https://freertos.org/Documentation/02-Kernel/03-Supported-devices/04-Demos/03-Emulation-and-simulation/Linux/FreeRTOS-simulator-for-Linux, 2025, accessed: 2025-12-15.
[54] E. ThreadX, "Posix compliancy wrapper for azure rtos threadx," https://github.com/eclipse-threadx/threadx/blob/master/utility/rtos_compatibility_layers/posix/readme_threadx_posix.txt, 2025, accessed: 2025-12-15.
[55] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.
[56] G. Bumble, "Bluetooth stack for apps, emulation, test and experimentation," https://github.com/google/bumble/, 2025, accessed: 2025-12-15.
[57] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, 2016, pp. 160–170.
[58] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 4, pp. 848–894, 1999.
[59] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.
[60] OpenSSL, "Openssl documentation," https://docs.openssl.org/master/, accessed: 2025-12-15.
[61] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.
[62] I. Angelakopoulos, G. Stringhini, and M. Egele, "Firmdiff: Improving the configuration of linux kernels geared towards firmware re-hosting," in *Proceedings of the NDSS Workshop on Binary Analysis Research (BAR)*, San Diego, CA, Feb. 2024.
[63] W. Li, L. Guan, J. Lin, J. Shi, and F. Li, "From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware," Feb. 2021.
[64] H. Green and T. Avgerinos, "Graphfuzz: library api fuzzing with lifetime-aware dataflow graphs," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, USA, May 2022.
[65] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu, "Apicraft: Fuzz driver generation for closed-source sdk libraries," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, Aug. 2021.
[66] R. OS, "Gnrc lorawan — riot documentation," https://api.riot-os.org/group__net__gnrc__lorawan.html, 2025, accessed: 2025-12-15.
[67] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise mmio modeling for effective firmware fuzzing," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.
[68] L. Project, "libfuzzer - a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html#corpus, 2025, accessed: 2025-12-15.
[69] Google, "Structure-aware fuzzing with libfuzzer," https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md, 2025, accessed: 2025-12-15.

APPENDIX A
ADDITIONAL CASE STUDIES

Case study 1: An out-of-bounds read in RIOT (Bug #20). Figure 9 shows an out-of-bound read vulnerability discovered by RTCON in RIOT. The vulnerability is found in `gnrc_lorawan_mic_is_valid` of RIOT LoRaWAN module [66]. The vulnerability is triggered at line 16; it reads the `fcnt` field of the `lorawan_hdr` structure without checking the size of the received buffer. Consequently, this leads to an out-of-bounds read in the user input buffer `buf`.

To reach the vulnerable code, we need to pass through device-specific functions (Lines 22-23) that are initialized during the boot process. Without proper initialization of these devices, traditional fuzzers would crash when attempting to call an uninitialized function (`dev->driver->set`) (Line 3) or reference an empty device structure (`&lw_netif->timer`)

```c
static void _sleep_radio(gnrc_lorawan_t *mac) {
    netdev_t *dev = gnrc_lorawan_get_netdev(mac);
    dev->driver->set(dev, NETOPT_STATE, &state, sizeof(state));
}

void gnrc_lorawan_remove_timer(gnrc_lorawan_t *mac) {
    gnrc_netif_lorawan_t *lw_netif = container_of(mac, ...);
    ztimer_remove(ZTIMER_MSEC, &lw_netif->timer);
}

static int gnrc_lorawan_mic_is_valid(uint8_t *buf, size_t len,
                                     ... ) {
    lorawan_hdr_t *lw_hdr = (lorawan_hdr_t *)buf;

    // No header size check before referencing the header
    uint32_t fcnt = byteorder_ltohs(lw_hdr->fcnt);
}

void gnrc_lorawan_radio_rx_done_cb(gnrc_lorawan_t *mac,
                                   uint8_t *psdu, size_t size) {
    // Device specific codes
    _sleep_radio(mac);
    gnrc_lorawan_remove_timer(mac);

    // Call to vulnerable function
    gnrc_lorawan_mic_is_valid(psdu, size);
}
```

Fig. 9: An out-of-bound read vulnerability in RIOT.

```c
static void rfcomm_handle_data(struct bt_rfcomm_session *session,
                               struct net_buf *buf, uint8_t dlci, uint8_t pf) {
    struct bt_rfcomm_dlc *dlc;

    // 1. Get Data Link Connection (dlc) object from current session
    dlc = rfcomm_dlcs_lookup_dlci(session->dlcs, dlci);
    if (!dlc) {
        LOG_ERR("Data recvd in non existing DLC");
        rfcomm_send_dm(session, dlci);
        return;
    }

    // 2. Check if the Data Link Connection is connected
    if (dlc->state != BT_RFCOMM_STATE_CONNECTED) {
        return;
    }

    // 3. Check Poll/Final bit in the frame
    if (pf == BT_RFCOMM_PF_UIH_CREDIT) {
        // net_buf_pull_u8 pulls data from the empty buffer buf
        rfcomm_dlc_tx_give_credits(dlc, net_buf_pull_u8(buf));
    }
    ...
}
```

Fig. 10: CVE-2024-6258: Out-of-bounds read in Zephyr.

(Line 8). This may be resolved by emulating the peripherals [38], [61], [67] or testing on real devices [41] equipped with corresponding peripherals. However, such approaches are time-consuming and known to have scalability issues. In contrast, RTCON can generate *mockup* device structures in the fuzzing campaign, allowing it to directly fuzz the target function without emulation or real devices.

Case study 2: Out-of-bounds read in Zephyr (Bug #7, CVE-2024-6258). Figure 10 shows an out-of-bounds read vulnerability in `rfcomm_handle_data` in Zephyr, which was discovered by RTCON. The vulnerability is triggered at line 21; `net_buf_pull_u8` can pull data from an empty buffer as this function does not check the remaining data length. Even though the bug seems to be simple, it is non-trivial to reach this code. For that, we require ① a valid Bluetooth RFCOMM session as well as Data Link Connection (`dlc`) (Line 6) and ② state of Data Link Connection should be in the connected state (Line 14). Additionally, ③ the specific bitfield of frame type (`pf`) should be set to `BT_RFCOMM_PF_UIH_CREDIT` (Line 19). As we can imagine, it is challenging to reach this code with end-to-end testing or traditional function-level fuzzing.

```c
// Global buffer size: 253
uint8_t rsp_buf[253];

static ssize_t ascs_qos(struct bt_conn *conn,
                        struct net_buf_simple *buf) {
    struct bt_ascs_qos_op *req;
    req = net_buf_simple_pull_mem(buf, sizeof(*req));

    // Vulnerable code
    // Missing bound check for req->num_ases
    for (uint8_t i = 0; i < req->num_ases; i++) {
        // Parsing QoS request
        qos = net_buf_simple_pull_mem(buf, sizeof(*qos));

        // Find Audio Stream Endpoint(ASE) by id
        ase = ase_find(conn, qos->ase);
        if (!ase) {
            // allocate response buffer with
            // size of 3 (sizeof(*ase_rsp))
            ase_rsp = net_buf_simple_add(&rsp_buf,
                                         sizeof(*ase_rsp));
            // Global Out-of-bounds Write
            ase_rsp->id = qos->ase;
            continue;
        }
    }
}
```

Fig. 11: CVE-2024-6442: Out-of-bound write in Zephyr.

In more detail, end-to-end testing requires a valid RFCOMM session, which is difficult to set up due to the complexity of the Bluetooth protocol. Similarly, traditional function-level fuzzing cannot reach the vulnerable code as it requires specific contexts that are hard to generate due to the complexity of their structures (e.g., `session`). However, RTCON can adaptively generate valid contexts to reach the vulnerable code effectively. This is achieved by populating appropriate context values on demand, without the need for analyzing complex structures.

Case study 3: Out-of-bound write in Zephyr (Bug #6, CVE-2024-6442). Figure 11 of Appendix shows an out-of-bound write vulnerability discovered by RTCON in Zephyr. The vulnerability is found in `ascs_qos` of Zephyr Bluetooth Audio Stream Control Service (ASCS). The vulnerability is triggered at line 11; it fails to check the maximum number of Audio Stream Endpoint (ASE) QoS requests (`req->num_ases`) before using it in the loop condition. This leads to QoS requests being excessively processed, each allocating an ASE id and adding it into the global response buffer `rsp_buf` (Line 20). Consequently, this results in an out-of-bounds write to the global buffer in the following code at line 23.

APPENDIX B
ROADBLOCK FUNCTIONS

Figure 12 shows an example of checksum validation in Zephyr, FreeRTOS, which led to 13 bugs being missed by the verifier harnesses. In Figure 12a, `rfcomm_recv` starts with receiving a message `buf` from the user. The function then validates the checksum of the message using `rfcomm_check_fcs` (Line 5). If the checksum is invalid, the function returns an error without reaching the vulnerable function `rfcomm_handle_data` (Line 11). Similarly, in Figure 12b, the checksum validation routines in IP packet processing, `usGenerateChecksum` (Line 1) and `usGenerateProtocolChecksum` (Line 9) immediately stop packet processing when the checksum is invalid. Therefore,

```c
static int rfcomm_recv(struct bt_l2cap_chan *chan,
                       struct net_buf *buf) {
    ...
    /* It should pass crc checksum */
    if (!rfcomm_check_fcs(fcs_len, buf->data, fcs)) {
        LOG_ERR("FCS check failed");
        return 0;
    }
    ...
    case BT_RFCOMM_UIH:
        rfcomm_handle_data(...); // target function
    ...
}
```

(a) A checksum validation routine in Zephyr

```c
if (usGenerateChecksum(0U,
    (const uint8_t *)&(pxIPHeader->ucVersionHeaderLength),
    (size_t)uxHeaderLength) != ipCORRECT_CRC)
{
    /* Check sum in IP-header not correct. */
    eReturn = eReleaseBuffer;
}
/* Is the upper-layer checksum (TCP/UDP/ICMP) correct? */
else if (usGenerateProtocolChecksum(
    (uint8_t *)(pxNetworkBuffer->pucEthernetBuffer),
    pxNetworkBuffer->xDataLength,
    pdFALSE) != ipCORRECT_CRC)
{
    /* Protocol checksum not accepted. */
    eReturn = eReleaseBuffer;
}
else
{
    /* The checksum of the received packet is OK. */
}
```

(b) A checksum validation routine in FreeRTOS

Fig. 12: An example of road-blocking routine

to reach the vulnerable function, the verifier harness should generate a valid message that passes the checksum validation. However, due to the complexity of the checksum algorithm, the verifier harness could not generate such a message within the limited time frame.

APPENDIX C
BUG FINDING CAPABILITY ON KNOWN RTOS VULNERABILITIES

Experimental Setup. We further evaluate RTCON on publicly disclosed vulnerabilities in three RTOSes: Zephyr, RIOT, and ThreadX. In total, we collected 20 vulnerabilities that are solely triggered by user inputs: 8 from Zephyr, 6 from RIOT, and 6 from ThreadX. In the case of FreeRTOS, we could not find any publicly disclosed vulnerabilities with sufficient information for reproduction. For each vulnerability, we set up its corresponding vulnerable version and executed RTCON for 24 hours under the same experimental environment as described in §VI.

Result. Table VII shows the detailed results of RTCON's bug finding capability on previously known RTOS vulnerabilities. To summarize, RTCON successfully detected all bugs except for CVE-2025-0727 in ThreadX. RTCON could not detect CVE-2025-0727 because it requires highly structured input (i.e., HTTP PUT message). Within the given time limit, RTCON could not generate such highly structured inputs (i.e., an HTTP request) from random bytes. Nevertheless, since RTCON generates harnesses based on libfuzzer [34], it can be addressed by supplying predefined corpora [68] and custom mutators [69] to effectively test structured inputs.

TABLE VII: Bug finding capability of RTCON on previously known RTOS vulnerabilities. Note that FreeRTOS vulnerabilities are excluded due to lack of disclosure. RTCON detected all bugs except CVE-2025-0727 in ThreadX, which requires highly structured input (e.g., HTTP PUT message).

Project	CVE	Subsystem	Found	Confidence		
				H	U	L
Zephyr	CVE-2021-3323	6LoWPAN	✔		✔	
	CVE-2021-3434	BT L2CAP	✔	✔		
	CVE-2021-3966	BT HCI	✔		✔	
	CVE-2023-0396	BT HCI	✔	✔		
	CVE-2023-4264	BT Audio	✔		✔	
	GHSA-56p9-5p3v-hhrc	MGMT	✔		✔	
	CVE-2023-5055	BT L2CAP	✔	✔		
	CVE-2024-3077	BT GATT	✔		✔	
RIOT	CVE-2023-24817	GNRC	✔	✔		
	CVE-2023-24819	6LoWPAN	✔		✔	
	CVE-2023-24820	6LoWPAN	✔		✔	
	CVE-2023-24821	6LoWPAN	✔	✔		
	CVE-2023-24825	GNRC	✔	✔		
	CVE-2023-33975	6LoWPAN	✔	✔		
ThreadX	CVE-2025-0727	HTTP	✗			
	CVE-2025-55090	IP	✔		✔	
	CVE-2025-55091	IP	✔		✔	
	CVE-2025-55092	IP	✔			✔
	CVE-2025-55093	IP	✔		✔	
	CVE-2025-55094	ICMP	✔			✔

Other vulnerabilities, all except CVE-2025-55092 and CVE-2025-55094, were detected with either high or unverifiable confidence, meaning that RTCON either verified the bugs from the top-layer function or detected them directly at the top-layer function. These two vulnerabilities could not be verified because their execution paths are blocked by checksum validation routines, which are the roadblock functions discussed in §B.

TABLE VIII: Detailed total line coverage measured in RTOSes. p-values (two-sided) are bolded when they show statistical significance ($p < 0.05$).

RTOS	RTCon	Manual			RTCon$^-$			FuzzSlice			FuzzGen		
	Mean	Mean	Difference	p-value	Mean	Difference	p-value	Mean	Difference	p-value	Mean	Difference	p-value
Zephyr	12.17%	**12.81%**	-0.64%	**0.008**	5.25%	6.93%	**0.012**	0.00%	12.17%	**0.007**	3.16%	9.02%	**0.007**
RIOT	19.96%	**20.81%**	-0.86%	0.222	11.84%	8.12%	**0.008**	5.99%	13.97%	**0.012**	3.92%	16.04%	**0.011**
FreeRTOS	**30.52%**	22.51%	8.02%	**0.008**	14.20%	16.32%	**0.012**	6.49%	24.04%	**0.011**	2.03%	28.49%	**0.012**
ThreadX	**11.74%**	11.45%	0.29%	0.310	7.23%	4.51%	**0.008**	1.76%	9.97%	**0.011**	0.52%	11.22%	**0.010**

```c
/* Adapter for testing net_buf structure */
void testNetBuf(__uint8_t *Data, size_t Size) {
  struct net_buf buf;
  buf.data = Data;
  buf.__buf = Data;
  buf.size = Size;
  buf.len = Size;
  /* Invoke the target function */
  fuzzEntryFunction(&buf, Size);
}

static void fuzzEntryFunctionHelper(__uint8_t *Data,
                                    size_t Size) {
  /* User can change the adapter function here */
  testNetBuf(Data, Size);
}
```

Fig. 13: Adapter code for testing `net_buf` in Zephyr.

TABLE IX: Bug finding capability of RTCon in general libraries, compared to AFGen.

Library	ID	Type	Found
libtiff	CVE-2022-34526	Buffer Overflow	-
ffjpeg	pull_request_45	Buffer Overflow	✔
	pull_request_46	Buffer Overflow	✔
	CVE-2021-45385	Buffer Overflow	✔
tcpreplay	CVE-2021-45386	Reachable Assertion	✔
	CVE-2021-45387	Reachable Assertion	✔
	CVE-2022-45484	Reachable Assertion	✔
	CVE-2023-27783	Reachable Assertion	✔
	CVE-2023-27784	Null Pointer Deref	✔
	CVE-2023-27785	Null Pointer Deref	✔
	CVE-2023-27786	Null Pointer Deref	✔
	CVE-2023-27787	Null Pointer Deref	✔
	CVE-2023-27788	Reachable Assertion	✔
	CVE-2023-27789	Reachable Assertion	✔
libming	CVE-2023-23051	Memory Leak	✔
	CVE-2023-23052	Memory Leak	✔
	CVE-2023-23053	Memory Leak	✔
	CVE-2023-23054	Infinite Loop	N/A
ngiflib	CVE-2021-36530	Buffer Overflow	✔
	CVE-2021-36531	Buffer Overflow	✔
	CVE-2022-30857	Buffer Overflow	✔
	CVE-2022-30858	Buffer Overflow	▲
liblouis	CVE-2023-26767	Buffer Overflow	✔
	CVE-2023-26768	Buffer Overflow	✔
	CVE-2023-26769	Buffer Overflow	✔
jhead	CVE-2022-28550	Buffer Overflow	✔

-: Not reproduced in RTCon or AFGen. ✔: Found.
▲: Found with simple hooks. N/A: Not applicable

TABLE X: Average harness size, analysis time of Analyzer, and build time with and without ACG for each RTOS. The analysis was conducted on the same machine used for the evaluation, using a single core, and repeated five times. Note that both Analyzer and ACG need to be executed only once per harness. The analysis time increases as the size of the target RTOS codebase grows (e.g., Zephyr) or when the target function becomes more complex (e.g., ThreadX).

	Size (MB)	Analyzer (Building CG) (s)	w/o ACG (s)	Taint analysis + ACG (s)
Zephyr	170.5	2146.0	149.1	274.4 (+84.0 %)
RIOT	7.1	86.1	27.9	64.1 (+129.7 %)
FreeRTOS	11.4	26.3	5.6	20.6 (+267.9 %)
ThreadX	38.9	234.6	107.2	221.8 (+106.9 %)

Algorithm 1: Single Function Taint Propagation

Input: Function F, Taint instruction set T, Context parameters $\{p_1, p_2, \ldots, p_n\}$
Output: Updated Taint instruction set T

Function `TaintPropagation`$(F,\ T,\ \{p_1, p_2, \ldots, p_n\})$**:**
  $T \leftarrow T \cup \{p_1, p_2, \ldots, p_n\}$;
  for $I \in F$ **do**
    if *I is Store(from, to)* **then**
      if *$from \in T$ and $to \notin T$* **then**
        $T \leftarrow T \cup \{to, I\}$;
      else if *$to \in T$* **then**
        $T \leftarrow T \cup \{I\}$;
    else if *I is (Load or GetElementPtr or Phi or Binary or Cast or Cmp or Switch)* **then**
      if *$\exists$ operand $\in$ operands s.t. operand $\in T$* **then**
        $T \leftarrow T \cup \{I\}$;
    else if *I is Call(callee, args...)* **then**
      if *isFunctionAnalyzed(callee)* **then**
        if *isReturnTainted(callee)* **then**
          $T \leftarrow T \cup \{I\}$;
        foreach *$idx \in$ taintedParamIdxs(callee)* **do**
          $T \leftarrow T \cup \{args[idx]\}$;
      else
        $T \leftarrow T \cup \{I\}$;
  return T;

APPENDIX D
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: The artifact is provided as a repository that contains the build environment, scripts, and binaries necessary for executing the fuzzing experiments. The artifact package can be accessed at: https://doi.org/10.5281/zenodo.17540919

2) Hardware dependencies: A machine with at least 8 CPU cores and 16 GB RAM is recommended for single-function tests. Multi-function tests can utilize up to 92 CPU cores and may require high-performance compute resources.

3) Software dependencies: Docker, Docker compose

4) Benchmarks: The evaluation was conducted against internal functions derived from RTOS subsystems (e.g., Bluetooth stack, TCP/IP modules). Configuration files specifying function lists and test parameters are located in the repository.

B. Artifact Installation & Configuration

The installation and configuration steps for the artifact can be found in the repository README. The artifact can be executed directly via Docker without additional dependencies.

C. Experiment Workflow

Two experiments are supported: (1) Single-function fuzzing (recommended for quick evaluation). (2) Multi-function fuzzing (for large-scale, resource-intensive evaluation). The high-level experimental workflow consists of four steps: 1) installing and configuring the build and evaluation settings, 2) building test harnesses for function-level fuzzing, 3) running the fuzzer, and 4) analyzing the results.

D. Major Claims

- (C1): RTCON enables function-level fuzzing of RTOS kernels to discover real-world vulnerabilities. This is proven by the experiment (E1) whose results are reported in Table III.
- (C2): RTCON classifies detected crashes based on confidence levels. This is proven by the experiment (E1 and E2) whose results are reported in Table VI.
- (C3): RTCON achieves effective line coverage whose results are reported in Fig 10. This is proven by the exeperiment (E2).

E. Evaluation

While the experiments can partially reproduce the results, full reproduction requires conducting large-scale experiments for 24 hours on the same machine configuration described in the paper.

1) Experiment (E1): [Single-Function Fuzzing] [1 compute-hour]: This experiment builds and executes harnesses for single function-level fuzzing to detect real-world bugs in RTOS kernels. Discovered bugs are automatically classified by their confidence level.

[Preparation] Build the corresponding Docker image and identify the target function to fuzz. A detailed list of available target functions can be found in: *eval/config/[project]-func-list-reduced.txt*.

[Execution] After building the Docker image, run the following command to build the harness:

```
docker run -it cgcc-[project]:latest \
[file location] [function] [test type] \
[MLC enable]
```

The harness can be executed in three modes:

- **Naive Run:**
  ```
  mkdir corpus && \
  ./host_bin/<target>-fuzz corpus/ \
  &>/tmp/result
  ```
- **Run Without MLC (e.g., 30s timeout):**
  ```
  /scripts/run_fuzz_single.py \
  <Function> 30
  ```
- **Run With MLC (Recommended, e.g., 300s timeout):**
  ```
  /scripts/run_fuzz_cross.py 300
  ```

[Results] Crash reports are stored under: *crash_dir/crash_[function_name]*. When multi-layer fuzzing (MLC) is enabled, summarized high-confidence crash reports are available in: *crash_dir/high_confidence*. Coverage data for each harness is recorded in: *coverage_dir/coverage_[function_name]*.

2) Experiment (E2): [Multi-Function Fuzzing] [24 compute-hour]: This experiment builds and executes harnesses for entire function sets to reproduce the line coverage RTCON discovered.

[Preparation] Build the corresponding Docker image and configure the evaluation setup. A detailed experiment parameters in the configuration file: *eval/config/config.py*. By default, the configuration enables up to 92 CPU cores, a 24-hour timeout, and MLC disabled.

[Execution] After loading the Docker image, execute the following command within the *eval/scripts* directory to launch a large-scale fuzzing campaign:

```
# Format:
# ./run_eval.py <Project> [Subsystem]

# Example: RIOT
./run_eval.py RIOT
```

[Results] To inspect the overall coverage, run the base container while mounting the output directory using the following command:

```
docker run --rm -it \
--volume=[out directory]:/out cgcc-base:latest
```

Inside the container, merge and generate a summary of the coverage report with the following command:

```
# Format:
# /scripts/view_coverage.py [project]

# Example: RIOT
/scripts/view_coverage.py RIOT
```