RT-Fuzzer: Task Driven Fuzzing of Real Time Operating System Firmware

Abraham A. Clements*, Abel Gomez Rivera*, Richard Jiayang†, Kirill Levchenko†,
Rick Kennell‡ and Gabriela Ciocarlie§
*Sandia National Laboratories – (aacleme, aogomez)@sandia.gov
†University of Illinois Urbana-Champaign – (rjliu3, klevchen)@illinois.edu
‡Purdue University – rick@purdue.edu
§The Cybersecurity Manufacturing Innovation Institute and Stevens Institute of Technology – Gabriela.Ciocarlie@cymanii.org

***Abstract*—Embedded systems are integral to modern society and are increasingly being attacked, necessitating improved techniques to identifying and mitigating vulnerabilities. Fuzzing has proven to be a useful technique for identifying vulnerabilities. Nevertheless, the complexity of embedded systems using real-time operating systems (RTOSes) has limited the ability to even observe their execution, much less effectively fuzz them. Re-hosting these systems' firmware in an emulator has emerged as a technique to solve challenges with inspectability and parallelizing fuzzing, but challenges remain for complex RTOS-based systems. We present RT-Fuzzer, a technique that leverages the modularization of RTOS-based embedded systems into tasks to simplify re-hosting and enable effective feedback-directed fuzzing of complex embedded systems. RT-Fuzzer creates a custom initialization for the RTOS and core services in the emulator and then starts only the target task(s) for fuzzing. This simplifies re-hosting and enables the fuzzing effort to be focused on a selected task. We illustrate this technique on an open source RTOS and a commercial PLC discovering and reporting vulnerabilities in both.**

I. Introduction

Embedded systems are integral to modern society, serving as the backbone of critical infrastructure across various sectors, particularly in manufacturing. These systems – *e.g.,* Programmable Logic Controllers (PLC), Process Automation Controllers (PAC) – are designed to perform dedicated control functions within larger mechanical or electrical systems. They are essential for applications such as industrial automation, robotics, and smart manufacturing. As reliance on these systems increases, it becomes imperative to address their cyber-security vulnerabilities, which can lead to operational disruptions.

Over the last couple decades, attacks on control systems by targeting their embedded systems highlight the challenges in protecting these systems. Specific examples include, TRISIS [1], Industroyer/Crashoverride [2], Industroyer2 [3], Blackenergy(1-3) [4]. Other attacks on embedded systems include vehicles [5] and satellite internet modems [6] that disrupted Internet services across Europe in 2022. In addition, internet-of-things (IoT) devices have been compromised to create large botnets [7]. These incidents show that embedded systems are being attacked, their effects have real impact.

In response to security challenges, fuzzing has emerged as a widely recognized best practice for discovering and addressing vulnerabilities in software. However, the application of fuzzing to embedded systems presents unique challenges that complicate its effectiveness. Factors such as hardware dependencies, limited visibility into system operations, and the complexities of instrumentation hinder the ability to conduct thorough fuzz testing. For example, traditional fuzzing techniques often rely on the ability to observe program execution. This requires hardware support on embedded systems, which may not be present, or should be disabled for security purposes. Consequently, the inherent characteristics of embedded systems necessitate innovative approaches to enhance the fuzzing process and improve vulnerability detection.

To address these challenges, researchers have proposed re-hosting [8], [9], [10], [11], [12], [13] firmware as a potential solution. This approach executes the firmware on a different host – typically an emulator –than the intended system. This allows the firmware to be executed in a more flexible and observable environment, facilitating more effective fuzzing methodologies. Significant work has been done in this area for Type 1[1] firmware that uses a traditional operating system (OS) (*e.g.,* embedded Linux). This includes Firmadyne [15] and other approaches [16]. Techniques for re-hosting and fuzzing Type 3 systems (bare-metal systems) are also becoming more common. This includes works like HALucinator [8], P2IM [10], DICE [17], Pretender [11], and others.

While these works show promise, there is limited work that focuses on Type 2 embedded systems that use either real-time operating systems (RTOSes) or custom-built embedded operating systems. Most controllers used in industrial-control-systems applications are Type 2 systems. Re-hosting such systems is particularly challenging because the firmware may be comprised of hundreds of tasks that have asynchronous

[1]Here we use the classification of embedded systems by Muench et al. [14]

Workshop on Binary Analysis Research (BAR) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-08-4
https://dx.doi.org/10.14722/bar.2026.23050
www.ndss-symposium.org

behavior and complex data dependencies between them. Some efforts to re-host and fuzz these systems includes work to re-host VxWorks [18], which describes a general process for replacing hardware-specific APIs with ones adapted to an emulator. This work addresses some of the challenges with re-hosting, but does not address how to fuzz the firmware. JetSet [9] uses concolic execution with a graph search algorithm to try and execute to a point where fuzzing data can be injected. Similarly, SFuzz [12] targets Type 2 systems by creating program slices from target functions to likely error-prone methods (*e.g.,* `memcpy`, `strcpy`) and applies localized fuzzing to these slices. SFuzz benefits from the modularization created by RTOS tasks, but it does not explicitly use them.

Our work adds to the body of work targeting Type 2 systems. We leverage the insight that, when fuzzing an embedded system, only the tasks that process the fuzzing input needs to be executed. Executing tasks that are not influenced by the fuzzing input is wasted execution. Thus, we shift our goal from re-hosting the entire firmware to building functional execution contexts for single tasks. We then fuzz only targeted tasks. This simplifies re-hosting and solves the issues with asynchronous and non-deterministic execution that prevents feedback-directed fuzzing from working properly.

This approach simplifies re-hosting since only the target task needs to execute, rather than the hardware and platform-specific RTOS mechanisms. It is also scalable, as the same initialized RTOS state can be used for multiple target tasks. Thus, many tasks in a firmware can be fuzzed after the creation of the initialization task. In addition, by enabling testing of individual tasks that developers are familiar with, our approach can be readily adopted by original equipment manufacturers (OEMs) in their testing environments.

In summary, this work provides the following novel contributions:

- A systematic way to create a base fuzzing context for Type 2 systems that enables targeted fuzzing of specific tasks;
- Extensions to an HALucinator [8] that enables it to use C code to instrument re-hosted firmware.
- A fuzzer, RT-Fuzzer, that leverages fuzzing contexts to fuzz commercial-of-the-shelf (COTS) firmware.
- The ability to move system state between an interactive re-hosting environment—which enables interactive inspection and debugging—and a hardware-independent fuzzing environment.

II. Background

Before describing the details of RT-Fuzzer, we discuss some preliminary background information.

A. Embedded System Types

Muench et al. [14] categorize embedded systems into Types 1, 2, and 3. Type 1 embedded systems adapt a general purpose OS to an embedded system (*e.g.,* Linux, RT-Linux). While at the other end of the spectrum are Type 3 systems – bare-metal systems or systems that use a very simple OS and often perform only one or two dedicated functions.

Sitting between Type 1 and Type 3 systems are Type 2 systems. These systems use a custom OS developed specifically for embedded systems, typically an RTOS. Examples include VxWorks, INTEGRITY, RTEMS, QNX, and Zephyr. These RTOSes frequently provide file systems, networking stacks, task synchronization methods, and other services. In a Type 2 system, all execution is organized in tasks which the kernel manages, typically using a priority-based scheduler.

Our work targets Type 2 systems. We expect these systems to be running dozens to hundreds of tasks, have clearly defined OS functionality, and—most importantly—execution of code in the system occurs in tasks. Each task has its own execution environment (including its own stack), but not necessarily memory space and its execution can be suspended and resumed by the kernel. We leverage the organization of firmware into tasks to simplify re-hosting by building fuzzing environments targeting specific tasks.

B. Firmware Fuzzing Challenges

Here, we briefly highlight specific challenges to fuzzing firmware and some common approaches to them. A more detailed review of related work is provided in section VI.

Parallelization of fuzzing. Embedded system firmware executes on custom-built computing systems and will not run on servers or desktops traditionally used for fuzzing. Thus, fuzzing is either performed on the original hardware or by re-hosting it in an emulator. Fuzzing on hardware limits parallelism (it requires potentially expensive hardware), limits visibility into the system's execution—preventing feedback directed fuzzing, and makes discovering and triaging bugs difficult. In addition, there is the risk of damaging the hardware (*e.g.,* bricking), if critical non-volatile data is corrupted.

Deterministic execution Modern fuzzing usually uses execution feedback to prioritize executing inputs that execute diverse code paths. For execution feedback to work properly, execution needs to be deterministic. That is, for a given input, the exact same code is executed in the same order. This allows mapping a input to code and inputs that execute the same code are minimized to prevent wasting execution time on testing the same code repeatedly. Type 2 systems interleave the execution of multiple tasks based off asynchronous events (*e.g.,* timer and communication interface interrupts), creating non deterministic execution. This prevents the fuzzer from minimizing its input set.

What to and how to fuzz. In a Type 2 system with many server-like tasks listening for inputs on network and other communication busses simply determining what to fuzz can be challenging. Further, these task may only start with a specific configuration and when running may not provide any external indication they are running until data is received. Sending data to them requires knowledge of the physical and logical ports to provide data to, along with some understanding of

the expected data formats. Further complicating this task—especially if using hardware—is that the physical interfaces on which to provide inputs vary greatly (*e.g.,* Ethernet, CAN bus, shared memory, custom backplanes). Thus, just sending fuzzing data can become a challenge.

Determining an error has occurred. The final challenges to fuzzing Type 2 embedded systems are determining that an error has occurred. With Type 2 systems, we do not have standardized crash dumps, the existence and extent of these capabilities are RTOS-specific. Furthermore, as described in [14], many memory corruptions, and other errors do not cause noticeable external events on embedded systems. Thus, we need means to instrument the system to detect crashes. Once we detect crashes, we need means to debug and inspect the system's execution to determine the root cause of the crash. This is complicated by asynchronous code execution and limited debugging capabilities.

C. Re-hosting Firmware

To overcome many of these challenges researchers and practitioners have turned to using re-hosting. Re-hosting provides nearly unlimited inspection capabilities and parallelizes across servers. The key challenge to re-hosting firmware is that the firmware is tightly coupled to the hardware and its peripherals. Due to the great diversity in hardware and peripherals, it is unlikely that any emulator will emulate all the required peripherals to execute a firmware out of the box. To address this, many approaches have been proposed, ranging from simply implementing the needed peripherals in the emulator, identifying and replacing abstraction layers in the firmware to decouple it from the hardware—also called high-level-emulation (HLE), to using machine learning (ML) approaches to provide inputs for the missing peripherals.

In this work, we use HLE as the structure of RTOSes makes this conducive. RTOSes are portable between different embedded systems by the use of a board support package (BSP), which abstracts the RTOS from the hardware it runs on. They also provide APIs for the creation and management of tasks, and frequently libraries implementing common abstractions like POSIX for file and network operations. Thus, for our design of executing a target task in a fuzzing context, the multitude of abstraction layers provides flexibility in how we enable the task to run in the emulator and natural places to instrument the firmware. In addition, by targeting common abstractions, our replacements for the RTOSes APIs can be reused, offering scalability for our solution. We will now explain the design and capabilities of RT-Fuzzer.

III. Design

Considering these challenges, we set out to design both a technique and a supporting proof-of-concept fuzzer called RT-Fuzzer that meets the following design goals:

1) Leverages firmware re-hosting to enable parallel fuzzing and provide inspect-ability during debugging.
2) Enables deterministic execution during fuzzing, allowing the use of feedback-directed fuzzing.
3) Enables fuzzing different inputs to the embedded system in a configurable manner.
4) Enables targeting specific tasks in the firmware to both focus fuzzing and reduce required understanding of the system.

Our design relies on the assumptions that: (1) the firmware can be unpacked to the executable code; (2) the memory layout and CPU architecture can be determined; and (3) an emulator that supports the CPU architecture is available.

We also assume the ability to apply function names to key API functions for initializing and creating tasks, and the API that we want to fuzz (*e.g.,* the socket library). In the worst case, these symbols can be found manually, and research in re-symbolizing [19], [20] software is continually decreasing the difficulty of labeling functions in binaries. Additionally, we aim to provide OEMs, who have symbols, new ways to assess their systems.

A. Overview

To meet our design goals, we use a key observation: the structuring of firmware into tasks provides the opportunity to target fuzzing to a specific task. Thus, we can simplify the re-hosting and fuzzing process by instead of re-hosting the whole firmware, re-hosting only a single task. In addition, if just a single task is running, and we do not emulate interrupts, we remove the asynchronous execution. However, tasks are not independent, they require system resources, and they interact with other tasks. Fortunately, they do this through APIs, for which we can use our re-hosting platform, HALucinator, to intercept. By intercepting and modeling the appropriate system resource APIs, functions that interact with other tasks, and functions that cause the task to yield, we can isolate the task and execute a single task, while still having it behave largely as it would in the real system. We call the context that isolates our target task a *fuzzing context*. This context simplifies re-hosting, provides deterministic execution, and enables the targeting of different input points in the firmware. Figure 1 illustrates the concept of fuzzing each task in its own fuzzing context.

To fuzz the an entire system, we break it into many fuzzing contexts and fuzz each one independently. For tasks that send data to other tasks, we envision capturing the APIs used to transfer data. Then, while fuzzing the sender task, we intercept the functions and fix up the state such that the execution can continue in the current context—*e.g.,* returning success to the sender like the message was successfully sent. We can also capture the data sent to the other task as potential inputs for a fuzzing corpus for the receiving task. The receiving task would then be set up in its own fuzzing context and be fuzzed independently, using the captured data as a seed corpus.

This approach is conceptually similar to Firmadyne [15] and FirmAE [16] that fuzz Linux-based firmware. They build a kernel for the emulator, then run programs from target firmware on their kernel. However, unlike on Linux systems, Type 2 systems often do not have well-defined separation between applications and system logic. The tasks are statically-linked with the kernel into a single binary and execute from

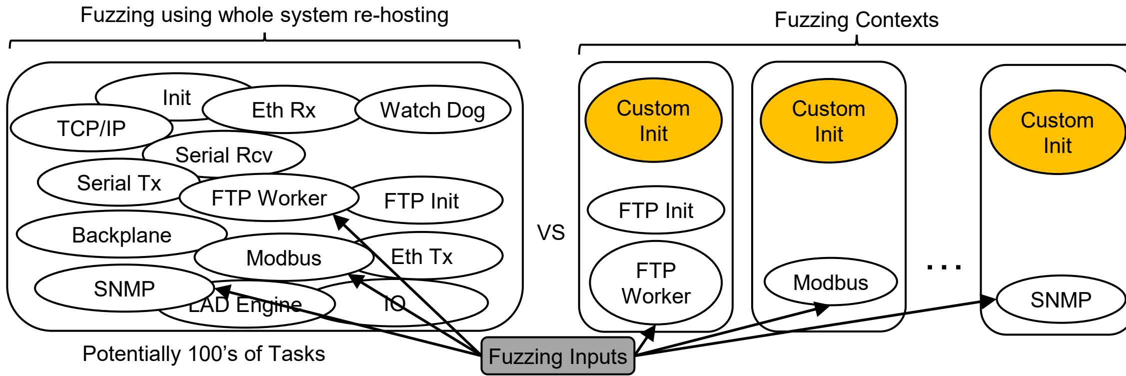

Fig. 1: Illustrates the conceptual idea of creating fuzzing contexts for fuzzing each task receiving fuzzing input.

the same memory space. Thus, the kernel and tasks must be initialized in place, and we must develop a strategy for isolating each task's execution.

As detailed in the subsections below, our RT-Fuzzer design addresses these challenges with a workflow organized in three phases, as shown in Figure 2. (1) Interactive initialization builds the fuzzing context for our target task and executes the task until it receives the fuzzing input. At this point, a snapshot of the system is taken. (2) Fuzzing loads the snapshot into the fuzzer and executes fuzzing test cases until the target objective(s) are found. This identifies inputs that require triaging. (3) Triaging is carried out by interactively loading the snapshot back into the emulator, along with the discovered input. This lets the user interactively explore the results of an input's execution.

B. Interactive Initialization

The goal of the interactive initialization is to build the *fuzzing context* for our target task and input function. We do this by initializing the kernel and then replacing the initial task with our implementation that builds the context to execute our target task. This is done through a series of steps.

Step 1. Start the kernel. We execute from the firmware's entry point to the start of the initial task. Prior to starting the initial task, the firmware sets up its global state (*i.e.,* zeros the .bss section, initializes the .data section, etc.) and sets up the kernel's task list and state. When the initial task starts, it means the scheduler and core features of the kernel are operational.

Step 2. Replace Initial Task. Typically, the initial task will initialize hardware peripherals, drivers, and core services. However, it initializes hardware and peripherals for the original hardware that are not present in our re-hosting environment. Thus, we will intercept the entry point for the initial task and replace it with our own implementation. Our implementation initializes core services that are needed. The details of what needs to be started depend on our target task, but common components include memory allocation utilities, file system(s), and semaphore libraries.

These services are usually provided with the RTOS and designed to be ported to different hardware. Where possible, we leverage implementations that minimize hardware interactions (*e.g.,* we use a RAM-based file system instead of Flash).

```
error_t initServer(){
    serverCfg = allocate(sizeof(serverCfg))
    initServerConfig(serverCfg)
    svrTaskId = osCreateTask(serverTask, serverCfg)
}
error_t serverTask(config){
    sock = socket()
    bind(sock)
    while(shouldRun){
        listen(sock)
        accept(sock)
        //setup workerConfig
        osCreateTask(workerTask, workerCfg)
    }
    // clean up
}
error_t workerTask(workerConfig){
    while (connection not closed){
        data = read(sock)
        process_data(data)
    }
}
```

Listing 1: Pseudo-code for threaded server on an RTOS.

Minimizing hardware interactions reduces the effort needed to re-host the system. Thus, by substituting implementations that have minimal hardware interactions (*e.g.,* only rely on the CPU and RAM), we simplify re-hosting. This approach also increases determinism because it removes hardware interactions and reduces non-volatile state (*e.g.,* the file system is not persistent so each fuzzing run starts with the same data).

When it is not possible to remove hardware interaction, we replace the functions used for porting with our own functions by intercepting execution at run-time. For example, the system clock is often initialized early and during initialization only needed to get through wait methods. Although we may not have the same clock peripheral, we can intercept the start, stop, and set functions and replace them with our own implementations that work with our emulated clock. While this effort is manual, we only need to implement our driver functions for our clock once for each RTOS. After getting the required core services working, we can start our task.

Step 3. Identify target task's initialization. For the target task to execute in similar fashion to the real system, we need to identify how it is initialized. Consider Listing 1, which shows how a typical initialization function creates a server task that then starts a worker task to handle each connection.

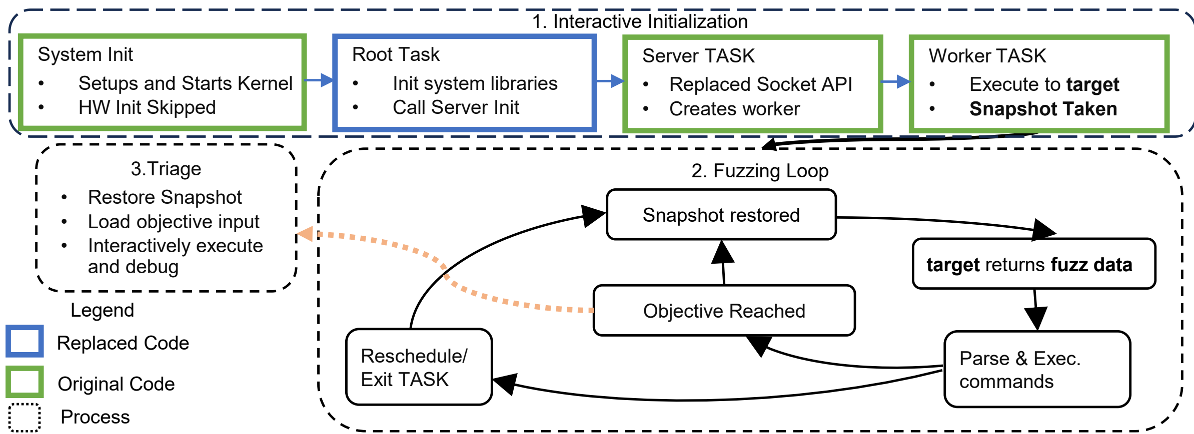

Fig. 2: Overview of fuzzing process for each context.

For fuzzing, we would want to inject the fuzzing data on the call to `read` in the worker task. However, if we just jump to the worker task, much of the server's state will not be set up. This results in immediate termination, or worse, many false positives while fuzzing. Therefore, we need to initialize the chain of tasks from the initialization function, through the server task to the worker task and into the worker task to the call to `read`.

To do this, we first identify the API function to which we will provide fuzzing input (*e.g.,* `read`). We then search the firmware for calls to this function using a software reverse-engineering tool (*e.g.,* Ghidra). Next, we search up the call tree to a task entry point. To find task entry points we find calls to the task create method (*e.g.,* `CreateTask`) using these functions. Tasks are created by calls to `CreateTask` from inside a task (*i.e.,* tasks create other tasks). Thus, to get the whole initialization state for a task, we need to identify the chain of tasks that execute prior to our call to our target function. To do this, we recursively identify the task entry points until we arrive back at the initial task. Ideally, this identifies a path of execution from system startup to our target task and input function. However, this relies on being able to create a complete inter-procedural call graph.

Fundamental limitations in binary analysis typically prevent recovering a complete call graph. Incomplete disassembly and unresolved function pointers are common causes of missing edges in the a call graph. When this occurs, we manually improve disassembly where possible, but another insight into embedded system firmware helps us with unresolved function pointers. Function pointers are typically used in code when multiple implementations are possible. If we cannot find a function that starts a task entry point, then we assume that a code pointer is used somewhere to start it and that there is not tight coupling between how the firmware starts the task, so we just start the highest level task we have found. We have also found that many server tasks start from an initialization function that takes no or only a few parameters. These are often task entry points, and we can build the needed context by executing these functions within our initial task.

After we identify the chain of tasks that call the function we want to fuzz, we then have our initialization task call this entry point when finished. In this way, we start the initialization process for our target task. However, execution may not arrive at our target task when started. The task likely only executes when it receives data and will wait on a semaphore or other barriers until data is received. For example, most TCP/IP servers will call the `listen` function, which will block until a connection is made, leading to Step 4.

Step 4. Intercept and replace needed system APIs with the ones suitable for fuzzing. The exact behavior for the API depends on the API, but the core idea is to force execution to our target function. For example, our implementation of `listen` always returns a socket that is connected and ready to be read from. In addition, semaphores are always available. In this way, execution will arrive at our target function.

At this point, we have the kernel started, have identified a task initialization process, and intercepted APIs to force execution to our function of interest. The next step is to execute to the point where our target function is called and then stop the emulator. Here, we take a snapshot of the entire system, saving the register and memory state.

C. *Fuzzing*

For fuzzing, we need the ability to load the snapshot, provide the fuzzing input, observe execution, determine when processing of the input is done, and detect crashes. Loading the snapshot is done by reversing the saving process. We perform it after each fuzzing input. By using an emulator, we can monitor execution of each basic block to enable feedback-directed fuzzing. To provide fuzzing input, we intercept the target function, and replace it with our own implementation that calls a special function `get_fuzz_data`, passing it a buffer and length. The fuzzer writes its data into this buffer prior to starting execution. When execution resumes, `get_fuzz_data` returns providing the fuzzing data.

Determining when execution of an input is done is more challenging. Embedded system server tasks typically run forever. Thus, we cannot just rely on the task exiting when it is done processing data. We have determined experimentally that input processing is usually complete when: an error function

is called, additional data beyond what was provided by the fuzzer is attempted to be read, the RTOS exits the task, or the scheduler tries to schedule another task. When we reach any of these points, we stop execution, restore the snapshot, and start a new fuzzing input.

In addition, to detect when processing of the input has completed, we need means to detect when the desired outcome of the fuzzing has been reached. While other objectives are possible the most common object is identifying vulnerabilities. Many of these can be detected by using the CPU's data abort exception indicating invalid memory was accessed.

D. Triaging Objectives and Limitations

Once objectives are found, it is often useful to interactively debug the system to understand which of them are true positives and which are false positives. In our design, we feed the fuzzing inputs back into our interactive emulation. We can then use a debugger and instrumentation in the emulator to understand what effects the input has on the system.

We are modifying the context in which the target task runs. That means that it may not execute as it would in the actual system, leading to our fuzzer having false positives and false negatives. This is true of all fuzzers, and this approach could lead to finding vulnerabilities that are not reachable in the real system today, but still lie dormant in the code base. From a defensive standpoint, mitigating these vulnerabilities could prevent problems if future updates make such code reachable.

IV. RT-Fuzzer Implementation

The implementation of RT-Fuzzer consists of common components and components that are customized on a per RTOS, firmware, or fuzzed task basis. We will first describe the common components and then use two firmware to highlight the implementation details of the remaining components.

A. Common Components

The common components only need implemented once. To implement RT-Fuzzer, we leveraged HALucinator and LibAFL [21].

1) Interactive Initialization: HALucinator is used for the interactive re-hosting to build the fuzzing contexts and to the take the snapshots used in fuzzing. It is used again in the triaging stage. LibAFL is used to build our fuzzer. This required extending both HALucinator's and LibAFL's capabilities. We extended HALucinator with the ability to compile C code into a binary that is loaded into the emulated environment. We then added the ability to overwrite functions in the original firmware to jump into the functions in our compiled binary. This allows us to replace functions in the firmware with our own implementations during run time. We also developed Ghidra scripts that allow exporting a header file and calling stubs that allow our compiled binary to call functions in the original firmware. These extensions to HALucinator are available in HALucinator's public repo [22]. We also extended LibAFL's QEMU's to be compatible with HALucinator allowing both to use the same version of QEMU.

For each fuzzed task we use these capabilities to build a custom program that is compiled and injected into the re-hosted system using the capabilities we added to HALucinator. This environment initializes the system, starts the target task, and intercepts needed API's to drive execution to the point fuzzing data is received by calling `get_fuzz_data`.

At this point, instrumentation is added to take a snapshot and exports a fuzzing configuration. The fuzzing configuration consists of the name of the snapshot to restore, a list of addresses indicating an objective has been reached (objective addresses) and another list of addresses indicating that fuzzing is done and the next fuzz run should start (done addresses).

2) Fuzzer: We implemented the fuzzer using LibAFL's QEMU system fuzzer [23]. It takes the snapshot and configuration. It loads the snapshot, and each fuzzing input is returned as the data returned from `get_fuzz_data`. Note that, because we added our instrumentation in a compiled program to the emulated system's memory, our instrumentation is captured in the snapshot. Thus, when the snapshot is restored, all needed instrumentation is in place. With the restored snapshot running and the fuzzing data input into the system, the fuzzer executes until a "done" address is reached, an objective address is reached, or a timeout occurs.

3) Triaging: Triaging is done by taking the inputs back into HALucinator and executing from the beginning to the `get_fuzz_data`. HALucinator is then used to provide the input from fuzzing as the data. Execution then continues and all the capabilities of HALucinator (debugger, python scripting, c-instrumentation) and QEMU's tracing capabilities are available for root cause analysis.

B. Target Components

The common components rely on components that are customized per target to enable re-hosting the firmware so it can be fuzzed. These components are: 1.) Starting the kernel and initialization task, 2.) Identifying the target task's initialization function and getting fuzzing data, 3.) drivers to abstract the API being fuzzed (e.g., socket library), and 4.) determining a when a crash has occurred or a fuzzing run is done. While, the components are custom per target, the use of common API's within and across firmware make it so the most of these components have the potential to be re-used across multiple firmware. We implemented these components for a Schneider Electric Modicon M340 PLC and an FTP server that ships with the open-source Apache NuttX [24] RTOS.

C. Modicon M340 PLC Example

1) Starting Kernel and Initialization Task: Starting the kernel required identifying the entry point and skipping a single function. Our initialization task replaces the firmware's original root task. It calls functions in the firmware to set up: memory, the math library, the input output (I/O) subsystem, semaphores, task variables, the clock library, and installs our custom socket driver into the I/O subsystem. It also calls the RTOS's provided functions to create a FAT file system using a RAM disk that remaps calls into the file system from using the

original file system. It then validates the RAM disk is setup properly by creating files and reading and writing to them.

This initialization task is common across the FTP Server, Modbus Server, and SNMP Server except how it starts the target task. The FTP and Modbus servers conclude the initialization task by calling the initialization function for the respective server. The Modbus Server required initializating it and skipping a call to file system initialization function. We use HALucinator to skip its execution. In total, the initialization tasks is 160 lines of C code. Two of which differ between the three fuzzed tasks.

2) Target Task Initialization and Getting Fuzz Data: We manually identified initialization procedures for the tasks by looking for relevant function names in the firmware's symbol table[2]. To find the fuzzing point we examined these tasks for calls to `socket` APIs and the locations that read data from the networks (*e.g.,* `recv`, `read`, `recvfrom`) using Ghidra and its identified "xrefs". Once these calls were identified, we manually traced xrefs back to the functions called to initialize the server and identified needed parameters for these calls.

3) API Replacement: With the initialization task starting our target task, we now need to avoid using hardware not present in the emulator, and receive fuzzing data. We do this by replacing the socket library with our own implementation. Our socket library, for the `socket` function, returns a socket descriptor that maps to our socket driver that the initialization task registered. Thus, calls to `open`, `close`, `write`, `ioctl` and, most importantly, `read` utilize our socket driver. We also replace `getaddrinfo`, `freeaddrinfo`, `gethostname`, `accept`, `bind`, `listen`, `select`, `getsockname`, and `connect` with our own implementations. These function's implementations return "success" and place constant data with sane values into structures expected to be populated by these functions. This allows the functions to execute without error and code using returned data to execute correctly.

Of particular importance for fuzzing is our implementations of `bind`, `listen`, and `accept`. The `bind` and `listen` calls always succeed and `listen` returns immediately, indicating that a connection has been made. `accept` also succeeds on the first call, returning a socket. Subsequent calls will take a semaphore. Thus, on the first call to `accept`, the code will execute as if a connection had been made and the server will create the worker task, providing it the socket and needed configuration. It will then block on the next call to `accept`, removing the server task from execution.

The kernel will then schedule the worker task. The worker task will eventually call `recv` or `read` depending on the server being fuzzed. In the target RTOS, `recv` looks up the I/O driver for the socket and calls `read`. Thus, both calls eventually call our `read` function that calls from `get_fuzz_data` getting data from either the fuzzer or interactively during triage.

4) Done Addresses and Detecting Objectives: We consider the fuzzing run to be done when 1) the scheduler removes the task from execution, 2) `get_fuzz_data` is called and all fuzzing data has been previously read, or 3) we hit functions indicating an error has occurred (*e.g.,* the input does not conform to the expected protocol and the code successfully detects the invalid input). These events all indicate the processing of the input in the target task has finished and the next input should execute. These addresses are set using either RTOS specific functions (e.g. `reschedule`) or discovered through experimentation. To detect objectives, we use the processor's exception vectors. Occurrence of exceptions indicate unexpected execution likely occurred that may lead to vulnerabilities.

D. Apache NuttX example

For NuttX, we used the built-in FTP server configured for a NUCLEO-H753ZI development board. This board has an ARM Cortex-M7 MCU with an Ethernet RJ45 port for network connectivity. It was built using the default build settings, with the included FTP server set to use default credentials, which includes an "anonymous" user that has no password. The default FTP server uses an in-memory file system. Additionally, we enabled stack canaries and debug assertions to catch more errors. This example shows that RT-Fuzzer can be used on binary firmware and also how it can be used by developers to identify and mitigate vulnerabilities.

1) Starting Kernel and Initialization Task: Starting the kernel required skipping clock configuration function after which the kernel started. We implemented our custom initialization by allowing the firmware to boot up as usual and start the FTP task. However, instead of spawning a new pthread for the task, we launch the task function synchronously. Removing the need for interrupts and context switching to start the task.

2) Target Task Initialization and Getting Fuzzing Data: These were identified by looking at the source and finding the appropriate function in the resulting binary. The FTP server calls `read` to receive data and our implementation calls `get_fuzz_data`. It also splits received data into multiple null-terminated strings to enables a single fuzz input to run multiple commands.

3) API Replacement: We used HALucinator to replace the socket library functions in the RTOS (`accept`, `bind`, `listen`, `connect`, and `socket`) and simply returns zero so that the RTOS believes it connected to a FTP connection and is ready to receive commands. `getsockname` returns a valid IP and port. The FTP server repeatedly calls `read` until the input is exhausted.

4) Done Addresses and Detecting Objectives : When the input data is exhausted we returned an empty string, causing the FTP server to close the connection. This allowed us to hook the `ftpd_close` function internally to indicate that the fuzzing was finished. We detected crashes by catching assertion failures, as well as the hardfault handler. Although assertion checks are typically not enabled in production firmware, their execution indicates a bug has likely occurred.

We've described implementations for a PLC and NuttX FTP server. Table I summarizes the setup requirements for apply-

[2]The binary is stripped but the RTOS embedded its own symbol table

Step	Scale Per	Notes
Initialization Task	RTOS	Small changes per target task.
Injecting Fuzz Data	Target Task	Identify call func. in task.
API Replacement	RTOS/Firm.	Write per RTOS, Id. per firmware.
Done Address	Target Task	Requires experimentation.
Objective Detection	Arch/RTOS	Arch. provides initial, RTOS APIs can enhance

TABLE I: Manual Effort Breakdown for RT-Fuzzer Setup

Task	Static Orig.	Static Instrumented	Fuzzing
FTP	17,445	14,860 (85.1%)	3,629 (20.8%)
SNMP	12,176	6,644 (54.5%)	1,270 (7.3%)
Modbus	16,672	10,520 (63.0%)	1,060 (6.3%)
NuttX-FTP	2,300	1,599 (69.5%)	2,474 (107%)

TABLE II: Reachability Analysis of basic blocks for each task

ing RT-Fuzzer. It distinguishes how each component scales with some scaling per RTOS, per firmware, and per target task being fuzzed. We observe that once the RTOS-specific components are implemented, the majority of the setup effort is in identifying the fuzzing target and done addresses. Our experience estimates this takes approximately 3–6 hours per task. Notably, the reusable components significantly reduces the effort for fuzzing multiple tasks within a firmware.

V. RT-Fuzzer Evaluation

To evaluate RT-Fuzzer, we designed experiments to examine how much code the fuzzer could explore, and how well it explored it. We also measured its ability to find bugs, execution speed, and evaluated its ability to triage findings. All experiments of RT-Fuzzer used our extensions to HALucinator, which extends LibAFL, and QEMU. Experiments were run on a server running Ubuntu 24.04, that has two Intel E5-2683v4 processors with 32 total cores, 512 GB of RAM and 750GB of storage. We provided input seeds to all fuzzing sessions, then the standard LibAFL mutator mutated the seed inputs throughout the fuzzing.

Target Firmware: For evaluation, we used the Modicon M340 firmware it runs on an ARM processor and uses a commercial RTOS. We specifically focused on three server tasks within the firmware: (1) FTP, (2) SNMP, and (3) Modbus. These tasks were also selected due to their widespread use in embedded systems and their potential for external network exposure. Each selected task within the firmware was encapsulated in its own *fuzzing context*, allowing us to isolate tasks and reduce the impact of asynchronous execution on the fuzzing process. We also evaluated the NuttX FTP server compiled for a NUCLEO-H753ZI development board.

To create the fuzzing context, we used HALucinator to run the firmware in a re-hosted environment. It was configured to run from the firmware entry point to our `RT_root` task that is modified for each experiment and concludes by calling the initialization function for each respective service.

A. RT-Fuzzer Reachability and Coverage Analysis

To evaluate how well we could fuzz each task, we needed to understand what code we could expect to be able to execute. To assess this, we performed a static reachability analysis on the original firmware and on the instrumented firmware. We then compare these static analysis to the coverage obtained dynamically during fuzzing. For static measurements, a custom Ghidra script was used to count the basic blocks that could be reached. For this analysis, we assumed, if a function could be reached, that all basic blocks in that function could be reached along with any outgoing function calls. We recursively counted basic blocks for each reachable function starting from the call to our target function until no new basic blocks were identified. Counting the basic blocks on the original code gives a lower bound of how many blocks could potentially be reached. It potentially under-counts blocks as it is not always possible to identify destinations of indirect calls.

It's important to compare the reachability on the original firmware to the instrumented firmware. To enable re-hosting, we replaced functions with our own versions by overwriting instructions in the firmware to jump to our implementations. Thus, our instrumented firmware modified the control flow with the potential to make significant parts unreachable and adds our own code. To account for this we performed the same reachability on the instrumented code, and removed any blocks we added from the count. This gives a measure of how much our instrumentation impacts reachability of the original code.

For example, our instrumentation of FTP reduced the number of blocks in the firmware by about 2,600 blocks as shown in Table II. Overall, our reachability showed that we reduced the expected executable code to between 54% and 85% of the original. This reflects the trade-off in fidelity of re-hosting: Higher fidelity emulation enables higher reachability. Here we traded ease of re-hosting—replacing functions (*e.g.,*. socket library)—at the cost of reduced reachability.

We ran three 24-hour fuzzing sessions for each task, using QEMU trace backends to log every executed basic block. As with static analysis, we excluded instrumentation-related blocks to isolate coverage within the original firmware. The "Fuzzing" column in Table II shows the union of blocks reached across the three runs. Fuzzing coverage varied significantly: FTP achieved 20.8% of static original blocks, while SNMP and Modbus reached only 7.3% and 6.3%, respectively.

Notably, Nutxx-FTP was able to dynamically execute more code than was discovered statically (107%). This is because handling FTP commands in the 'send' subtask, is driven by runtime conditions and indirect control flow, which static analysis cannot follow. This highlights the value of dynamic fuzzing in uncovering behavior that static analysis cannot.

B. Fuzzer Performance

To understand the fuzzer's ability to fuzz the systems and find objectives, we ran the fuzzer for 24 hours again with tracing *disabled* and averaged results over three runs. A summary of these results is presented in Table III. The *Corpora* column is the number of fuzzing inputs created during the fuzzing session. The number of corpora represents the total number of corpus inputs the fuzzer was continuing to mutate at the end of fuzzing. Larger numbers indicate the fuzzer found additional execution paths as only inputs that

Task	Corpora	Objectives	Executions	Exec/sec	Timeout	Exceptions
FTP	108.2k	3,855k	155.3M	3.2k	3,254k	601k
SNMP	28.5k	1,030	32,790M	94.96k	1,030	0
Modbus	4,134	5,432	9,533M	111.3k	5,432	0
NuttX-FTP	35,3k	9,971	1,288M	44.76k	9,971	0

TABLE III: Fuzzing Metrics of 24 hrs fuzzing session

executed unique code paths were kept. In each case, the fuzzer was able to find new code paths than the those executed by the original input seeds. However, complexity of SNMP and Modbus limited the exploration of these protocols. This is also reflected by the low coverage in Table II for these protocols. To improve coverage: the seed corpus could be expanded, the fuzzer could be made semantics-aware, or the fuzzer could be augmented with symbolic execution [25]. Our goal was to demonstrate the ability to fuzz individual tasks in Type 2 firmware, and the fuzzer's discovery of new paths shows that adopting additional techniques would only improve this.

Objectives are the number of inputs that triggered a processor exception or timeout. *Executions* show total inputs executed, and *Exec/sec* are the average number of executions per second. *Timeouts* occurred when the execution of QEMU did not return control to the fuzzer after one second. This long delay was the cause of FTP's lower executions per second. Its performance could have been improved by decreasing the timeout or identifying addresses where the timeouts occurred and adding them to the done addresses to start the next test case. *Exceptions* occurred when either a data-abort or invalid instruction exception occurred, likely indicating a bug.

C. Modicon M340 Findings

On the FTP process, all exceptions originating from the same code location. We selected a crashing input and loaded the snapshot backup into HALucinator and provided the input and interactively executed until the exception occurred. From this, it was apparent user input to the FTP server was not properly validated leading to a potential denial of service attack on the system. To further show the utility of RT-Fuzzer, we patched the bug by manually editing the binary. We then re-ran the fuzzer with the crashing inputs and original corpus and verified no exceptions occurred.

The bug and suggested patches were reported to Schneider Electric who have issued a security notification [26] and CVE [27]. The vulnerability received a CVSS v4.0 score of 8.7 and affects all firmware versions of all Modicon M340 processors and multiple versions of five related product lines.

To highlight the powerful capabilities of RT-Fuzzer consider using a physical system to find this bug. This would require the hardware, the software to configure the device, and creating a custom configuration to start the FTP server (it does not start by default). Detecting the exception is further complicated as the RTOS catches the exception resetting the device without the specific location or cause being captured. These challenges perhaps indicate why this bug has been present across all versions of the firmware. By directly starting the FTP server, using re-hosting, our approach makes locating the cause of the crash straightforward. This greatly simplifies fuzzing the system and identifying the bug.

D. Apache NuttX findings

We identified two issues in NuttX's internal file system. These bugs may be triggered by the numerous other interfaces to the NuttX, and are not limited to FTP. In this regard, FTP is merely used as an interface to the file system. The root issue of the first is de-referencing a NULL pointer in an inode. In debug builds an assert catches this, however, in release builds assertions are not present. This results in potential to corrupt an inode to pointer such that its child node points to itself, resulting in an infinite loop and a potential denial-of-service. The second bug is a use-after-free bug and has potential to allow altering unexpected memory. Both bugs and patches were reported to the NuttX developers. The patches were applied and CVES issued [28], [29].

E. Comparison with SFuzz

To understand the strengths and limitations of RT-Fuzzer, we compared it with SFuzz [12], a state-of-the-art fuzzing framework tailored for RTOS firmware. SFuzz integrates forward and backward slicing to isolate sensitive code paths and applies hybrid fuzzing—combining coverage-guided fuzzing with symbolic execution—to enhance path exploration. SFuzz reported the discovery of 77 previously unknown vulnerabilities across 35 RTOS samples.

We attempted to replicate SFuzz's results, but we encountered significant challenges. The original environment depended on outdated components, and several packages required replacement or patching to function with current toolchains. After updating we were able to validate the key steps it performed worked—specifically slice identification, creation of slices for fuzzing, and fuzzing. Despite adapting the setup and doubling the fuzzing time to 48 hours (compared to the 24 hours used in SFuzz experiments), we were unable to reproduce any of the vulnerabilities reported in the original paper. This suggests that SFuzz's effectiveness is highly sensitive to environmental factors.

In an effort to compare to RT-Fuzzer, we also ran SFuzz on the Modicon M340 firmware. SFuzz identified four slices it could fuzz in tasks related to FTP and DNS. Some of the slices are within the FTP server tasks although they do not overlap with basic blocks executed in the FTP worker task fuzzed using RT-Fuzzer. This is because slices between source methods (recv, read) and sensitive sink functions (e.g., memcpy) were not found. We ran SFuzz's fuzzing engine for 24 hours on the discovered slices. The fuzzer detected crashes during this phase, but these crashes were only valid in the instrumented binaries created by SFuzz to target the slices and could not be reproduced on the unmodified firmware. This discrepancy stems from path modifications introduced by SFuzz's instrumentation and slicing strategy. As a result, the crashes were artifacts of altered execution rather than genuine vulnerabilities in the target binary.

To further compare the approaches, we performed a basic block reachability analysis using both SFuzz and RT-Fuzzer. Table IV reports the results for SFuzz's four slices, showing the number of reachable blocks in the original binary versus

Task	Original	Fuzzing
Slice_1	7,329	190 (2.5%)
Slice_2	5,776	451 (7.8%)
Slice_3	5,776	479 (8.2%)
Slice_4	7,329	205 (2.8%)

TABLE IV: Basic block reachability in original firmware vs. SFuzz coverage

the number actually exercised during fuzzing. While SFuzz's slicing improves focus on specific code regions, its dynamic coverage remained low—under 10% in all cases. This limited reachability is consistent with SFuzz's design goal: restrict execution to only those paths considered relevant to a particular vulnerability slice. In contrast, RT-Fuzzer it restricts execution between tasks but allows full exploration *within* the target task. This results in higher overall code coverage and a better chance of discovering emergent behaviors or vulnerabilities that span beyond narrowly defined slices.

F. Evaluation Summary

We have demonstrated the ability of RT-Fuzzer to find real world bugs for a commercial PLC and popular open-source RTOS. This was achieved by building fuzzing contexts to enable targeted fuzzing of specific tasks within the firmware. We've also shown that the same initialization can be used to create fuzzing contexts for multiple tasks. Thus, the manual effort needed to create it has potential for extensive reuse. Finally, we've demonstrated how interactive analysis can evaluate and test patches enabling remediation of bugs.

VI. Related Work

Fuzz testing has emerged as a critical technique for identifying vulnerabilities in software, particularly in the context of embedded systems and firmware. The unique characteristics of embedded environments—such as tight integration with specific hardware, limited resources, and the absence of advanced debugging capabilities—pose significant challenges for effective vulnerability detection. As a result, researchers have developed a diverse array of fuzzing frameworks and methodologies tailored to address these challenges.

FT-Framework [30] employs a “forced execution” technique to fuzz incomplete firmware binaries, identifying executable function fragments without requiring complete binaries or source code. Similarly, Srivastav et al. [31] introduce ”tripwiring” to improve directed fuzzing by eliminating unreachable paths, while SDFUZZ [32] focuses on reducing unnecessary exploration by targeting specific states derived from vulnerability descriptions. Enhancements in binary-only fuzzing are also evident in ZAFL [33], which achieves near compiler-level efficiency, and FUZZWARE [34], which utilizes fine-grained memory-mapped I/O modeling to improve code coverage. Furthermore, SAFIREFUZZ [35] showcases a high-performance rehosting and fuzzing framework for ARM Cortex-M firmware, achieving a remarkable increase in fuzzing throughput and code coverage through near-native rehosting techniques. EM-Fuzz [36] further contributes to the field by combining fuzzing with real-time memory checking, significantly improving vulnerability detection in firmware and addressing the challenges of identifying memory vulnerabilities without source code.

Additionally, the complexities of fuzzing embedded systems have been extensively reviewed. Prior research [37], [13] categorizes fuzzing tools based on their interaction with embedded environments, highlighting the unique challenges and limitations these systems present. A comparative analysis [38] highlights the benefits and limitations of real-device-based versus simulation-based fuzzing approaches. The role of coverage feedback in bug discovery is critically examined in [39], revealing that its effectiveness can vary based on initial seed quality and system architecture. Moreover, the unique difficulties of fuzzing embedded systems, particularly regarding memory corruption, are addressed by Muench et al. [14]. Collectively, these contributions show the need for specialized fuzzing techniques for embedded systems.

Our approach to evaluate Type 2 embedded systems introduces several innovative features that set it apart from existing methodologies. We specifically address the unique challenges posed by these systems. SFuzz [12], as covered in our evaluation, targets fuzzing RTOSes by creating targeted slices of the firmware. One of the key innovations in our methodology is the introduction of *fuzzing contexts*, which allows for the independent fuzzing of individual tasks. This enhances determinism and simplifies the fuzzing process. It also allows our interactive initialization phase that distinguishes our approach by customizing the system state before fuzzing begins—something that existing tools like SFuzz, ZAFL, and EM-Fuzz do not emphasize. Our triaging phase allows for interactive debugging, enabling us to differentiate between true positives and false positives. This level of interactivity and insight into execution is often lacking in other frameworks.

VII. Discussion and Conclusion

Our ultimate goal is to improve the safety and security of embedded systems on which we all rely. The proposed approach enables both third parties and OEMs to improve their testing of firmware. This work shows how to decompose firmware into components to simplify re-hosting and enable effective parallel fuzzing. This decomposition along task boundaries is already used in development making the foundations of our approach readily approachable for developers.

We have shown that RT-Fuzzer can effectively enable fuzzing of real firmware and that a common custom initialization routine can be used to fuzz multiple server tasks in a firmware. Its utility was demonstrated finding three real-world bugs while fuzzing four tasks across two RTOSes.

Acknowledgment

This material is based upon work partially supported by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Advanced Materials and Manufacturing Technologies Office, Award Number DE-EE0009046.

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under contract DE-NA0003525. This written work is authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan. SAND2026-18021C

REFERENCES

[1] Dragos, "Trisis malware: Analysis of safety system targeted malware," Dragos, Inc, Tech. Rep., 2017. [Online]. Available: https://www.dragos.com/wp-content/uploads/TRISIS-01.pdf

[2] A. Cherepanov, "Win32/industroyer: A new threat for industrial control systems," ESET, Tech. Rep., 06 2017. [Online]. Available: https://web-assets.esetstatic.com/wls/2017/06/Win32_Industroyer.pdf

[3] E. Research, "Industroyer2: Industroyer reloaded." [Online]. Available: https://www.welivesecurity.com/2022/04/12/industroyer2-industroyer-reloaded/

[4] US-CISA, "Ongoing sophisticated malware campaign compromising ics (update e)," US Cybersecurity and Infrastructure Security Agency, Tech. Rep., 7 2021. [Online]. Available: https://www.cisa.gov/news-events/ics-alerts/ics-alert-14-281-01e

[5] A. Greenberg, "Hackers remotely kill a jeep on the highway—with me in it," *Wired Magazine*, 7 2015. [Online]. Available: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[6] Viasat, "Ka-sat network cyber attack overview," 2022. [Online]. Available: https://news.viasat.com/blog/corporate/ka-sat-network-cyber-attack-overview

[7] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[8] 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 *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1201–1218. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[9] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 321–338. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/johnson

[10] B. Feng, A. Mera, and L. Lu, "{P2IM}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1237–1254.

[11] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vinga, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 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 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 485–498. [Online]. Available: https://doi.org/10.1145/3548606.3559367

[13] J. Yun, F. Rustamov, J. Kim, and Y. Shin, "Fuzzing of embedded systems: A survey," *ACM Comput. Surv.*, vol. 55, no. 7, Dec. 2022. [Online]. Available: https://doi.org/10.1145/3538644

[14] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *Network and Distributed System Security Symposium*, Feb 2018.

[15] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware," in *23rd Annual Network and Distributed System Security Symposium, 2016, San Diego, California, USA, February 21-24, 2016*, 2016. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/towards-automated-dynamic-analysis-linux-based-embedded-firmware.pdf

[16] 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 36th Annual Computer Security Applications Conference*, 2020, pp. 733–745.

[17] A. Mera, B. Feng, L. Lu, and E. Kirda, "Dice: Automatic emulation of dma input channels for dynamic firmware analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1938–1954.

[18] A. A. Clements, L. Carpenter, W. A. Moeglein, and C. Wright, "Is your firmware real or re-hosted?" in *Workshop on Binary Analysis Research (BAR)*, vol. 2021, 2021, p. 21.

[19] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4554–4568.

[20] "Introduction to bsim," 2024. [Online]. Available: "https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/GhidraClass/BSim/BSimTutorial_Intro.md"

[21] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, ser. CCS '22. ACM, November 2022.

[22] "Halucinator source code." [Online]. Available: https://github.com/halucinator/halucinator

[23] "Libafl qemu bare-metal fuzzer," 2024. [Online]. Available: https://github.com/AFLplusplus/LibAFL/tree/main/fuzzers/full_system/qemu_baremetal

[24] "Apache nuttx," The Apache Software Foundation, 2025, real-time operating system for embedded systems. [Online]. Available: https://nuttx.apache.org/

[25] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[26] Schneider Electric, "Schneider electric security notification sevd-2025-224-05," 2025, accessed: 2026-01-05. [Online]. Available: https://download.schneider-electric.com/files?p_Doc_Ref=SEVD-2025-224-05&p_enDocType=Security+and+Safety+Notice&p_File_Name=SEVD-2025-224-05.pdf

[27] Schneider Electric SE, "Cve-2025-6625," 2025, accessed: 2026-02-18. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2025-6625

[28] Apache Software Foundation, "Cve-2025-48768," 2025, accessed: 2026-02-18. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2025-48768

[29] ——, "2025-07-07," 2025, accessed: 2026-02-18. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2025-48769

[30] J. Jang, G. Son, H. Lee, H. Yun, D. Kim, S. Lee, S. Kim, and D. Jang, "Fuzzability testing framework for incomplete firmware binary," *IEEE Access*, vol. 11, pp. 77 608–77 619, 2023.

[31] P. Srivastava, S. Nagy, M. Hicks, A. Bianchi, and M. Payer, "One fuzz doesn't fit all: Optimizing directed fuzzing via target-tailored program state restriction," in *Proceedings of the 38th Annual Computer Security Applications Conference*, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 388–399. [Online]. Available: https://doi.org/10.1145/3564625.3564643

[32] P. Li, W. Meng, and C. Zhang, "SDFuzz: Target states driven directed fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 2441–2457. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/li-penghui

[33] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1683–1700. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy

[34] 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 *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1239–1256. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[35] L. Seidel, D. C. Maier, and M. Muench, "Forming faster firmware fuzzers." in *USENIX Security Symposium*, 2023, pp. 2903–2920.

[36] J. Gao, Y. Xu, Y. Jiang, Z. Liu, W. Chang, X. Jiao, and J. Sun, "Em-fuzz: Augmented firmware fuzzing via memory checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3420–3432, 2020.

[37] M. Eisele, M. Maugeri, R. Shriwas, C. Huth, and G. Bella, "Embedded fuzzing: a review of challenges, tools, and solutions," *Cybersecurity*, vol. 5, no. 1, p. 18, 2022.

[38] C. Zhang, Y. Wang, and L. Wang, "Firmware fuzzing: The state of the art," in *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, ser. Internetware '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 110–115. [Online]. Available: https://doi.org/10.1145/3457913.3457934

[39] D. Jang, J. Kim, J. Kim, W. Im, M. Jeong, B. Choi, and C. Kil, "On the analysis of coverage feedback in a fuzzing proprietary system," *Applied Sciences*, vol. 14, no. 13, 2024. [Online]. Available: https://www.mdpi.com/2076-3417/14/13/5939