(Pre-Print) Is Your Firmware Real or Re-Hosted?
A case study in re-hosting VxWorks control system firmware

Abraham A. Clements
Sandia National Laboratories
aacleme@sandia.gov

Logan Carpenter
Sandia National Laboratories
lcarpen@sandia.gov

William A. Moeglein
Sandia National Laboratories
wmoegle@sandia.gov

Christopher Wright
Purdue University
christopherwright@purdue.edu

***Abstract*—Emulating firmware is increasingly popular for systems research, particularly vulnerability research. In this paper we describe how we extend HALucinator to work with real-world systems that use the popular VxWorks RTOS. We describe the Re-hosting Support Layer (its definition and implementation) with the functions necessary to get a Schneider Electric SCADAPack 350 remote terminal unit, a Schneider Electric Modicon 340 programmable logic controller, and Hughes 9201 BGAN inmarsat terminal up and re-hosted (at least partially). We share the process and our path of performing this work over the last year, and give a retrospective approach for re-hosting other RTOSes. We provide a case study with 3 real devices, and show that we can re-host portions of the firmware and perform analyses to show the success of our approach.**

I. Introduction

Firmware running on embedded systems is ubiquitous. Any modern system performing control or automation functionality likely contains an integrated micro-controller executing firmware. This firmware controls how the device behaves and interacts with the physical world. While we are dependent on these systems, analyzing firmware for security vulnerabilities is a challenging task because of its tight integration with hardware. This requires nearly all dynamic analysis of firmware to be performed executing the firmware on the hardware, which significantly limits the ability to inspect the its execution. Execution inspection capabilities are typically provided by a debugging port on hardware and are usually limited to a handful of breakpoints. Further, leaving the debugging port accessible is generally considered a poor security practice and thus is often not even an option for third parties trying to analyze firmware. Requiring hardware also increases the expense and decreases the scale at which automated analyses can be applied, e.g., programmable logic controllers can each cost several thousands of dollars. Automated vulnerability testing such as fuzzing requires running many instances of the device to enable effective discovery of vulnerabilities and the dollar cost of devices can significantly limit the scale at which testing can be performed.

Emulating embedded systems by re-hosting their firmware has been demonstrated as a technique to overcome the requirement of hardware in performing dynamic analysis of firmware [7], [10], [11], [27]. To do this, a firmware is run on a software emulator, which allows it be run on commodity desktops and servers. This also allows many instances of the system to be started – with the number of instances only limited by the availability of general purpose computing resources.

Workshop on Binary Analysis Research (BAR) 2021
February 2021, San Diego, CA, USA
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/bar.2021.23xxx

The primary challenge to re-hosting firmware is providing valid inputs for the hardware that is not implemented in the emulator. Emulators such as QEMU [6] provide the ability to emulate a variety of CPU architectures, but provide very limited support for the huge variety of peripherals (e.g., timers, UARTs, Ethernet controllers) used in commodity embedded systems. Many different approaches have been proposed to address these challenges. One approach is to implement the hardware at low-level memory mapped register interface as done in QEMU – a laborious task that does not scale well. Another approach is hardware-in-the-loop emulation [18], where accesses to peripherals are forwarded to physical hardware – which reduces its scalability. Machine learning [14] is an approach where interactions with peripherals are recorded on hardware and used to build models of peripherals. These models are then used during emulation. Another approach is to use a fuzzer [12] to provide inputs for the peripherals. Both machine learning and fuzzing peripherals limits control over the devices making them unsuitable for high fidelity emulation.

A promising approach is High Level Emulation (HLE) [8], [7], [10], [11] where common abstractions within the firmware are utilized to remove the need to provide low-level support for peripherals. Both Firmadyne [7] and Costin et al. [10], [11] utilize the Linux kernel abstractions to enable re-hosting Linux based firmware. HALucinator [8] uses hardware abstraction layers provided by micro-controller manufacturers, enabling re-hosting of simple bare-metal applications.

In this paper, we extend HALucinator to enable its use in re-hosting Real-Time Operating Systems (RTOS). For a more general review of re-hosting we refer the reader to [34]. RTOSes aim to make the process of writing firmware for embedded systems easier. These RTOSes provide the basic constructs for a real-time system and give the developer abstract mechanisms to define system behavior. These mechanisms are often organized as layers, which hide the working details of the system from higher layers. In addition, these operating systems define an interface–called a Board Support Package (BSP)–to enable portability across a large variety of hardware. These layers, and in particular the BSP, provide a natural place for HALucinator to decouple firmware from its hardware and enable re-hosting in an emulator.

To demonstrate this, we focus on VxWorks, a commercial RTOS commonly found in safety-critical industries such as aerospace, automotive, medical, and manufacturing [2], [15], [3]. It is used extensively by major companies including Siemens, Boeing, Bosch, Huawei, Northrop Grumman, and

others. WindRiver, the maker of VxWorks, estimates there are on over 2 billion devices running VxWorks [25]. Probably the most publicized use of VxWorks was in the Mars Rover, where NASA Jet Propulsion Laboratory (JPL) used VxWorks in the Mars Exploration Rover [30]. VxWorks has also been demonstrated to have critical vulnerabilities [4], [26].

By leveraging and replacing some of the abstraction layers used for the development of VxWorks, we have found that we can enable HALucinator to re-host the firmware from these complex systems. We refer to our abstraction replacements as a re-hosting support layer (RSL) and it enables file system operations, execution of multiple asynchronous tasks, recording system logs, and interactive communication over both Ethernet and serial ports. We demonstrate our re-hosting support layer, performing a set of dynamic analyses on firmware from a Schneider Electric SCADAPack 350 remote terminal unit, a Schneider Electric Modicon 340 programmable logic controller, and Hughes 9201 BGAN inmarsat terminal. We report our task analysis, initialization analysis, function/symbol analysis and execution statistics in section IV.

While the re-hosting support layer presented in this work is specific to VxWorks, we believe the concepts and lessons learned in its development are generally applicable to other real-time operating systems. Thus, we also provide a retrospective approach for others to use in development of re-hosting support layers for additional RTOSes. In summary this work presents using HALucinator to provide a re-hosting support layer for VxWorks to enable emulation of 3 real-world devices, analysis, detailed explanation, and statistics recorded while re-hosting these devices, and a retrospective and advice on re-hosting embedded RTOS devices.

II. Background

Before describing our re-hosting support layer, we formally define some of the terminology used in this work. We then discuss the benefits of re-hosting and the scale we use for determining the utility of a re-hosted firmware. Next, since our work requires a functional knowledge of both HALucinator and VxWorks we provide a brief review of each.

A. Vocabulary

Firmware Re-hosting: The process of executing firmware on a platform other than its originally intended hardware.

Emulation vs Re-hosting: The terms emulating a firmware and re-hosting firmware are often used interchangeably. This is because the two ideas are so closely related. In this paper, we refer to emulating a system by re-hosting a firmware. That is, the goal is to emulate the system for a specific purpose and the means of doing it is re-hosting the firmware. We find using this terminology helps making decisions during re-hosting, because the right decision usually depends on what we are trying to emulate. For example, do we need to provide high fidelity for power management in our re-hosting? If we are emulating to test power management, then yes; otherwise, probably not.

Re-hosting Support Layer (RSL): The functionality that is replaced during a firmware's execution to enable re-hosting. In the context of HALucinator and this paper, the re-hosting support layer is the set of handlers and models that enable HALucinator to re-host a firmware.

VxWorks vs Device Code: In developing a re-hosting support layer, it is often useful to distinguish the origin of the functionality being replaced. The origin of the code impacts how widely used the functionality is. Intercepting code that is part of VxWorks will be common across all systems using VxWorks and replacing it will enable wide application of the re-hosting support layer. On the other hand, device specific code will only be found on a device or a small family of devices. Replacing it limits applicability of the re-hosting support layer, but can increase fidelity of the emulated system. We will refer to code that is part of VxWorks as **VxWorks code** and all other code as **device code**.

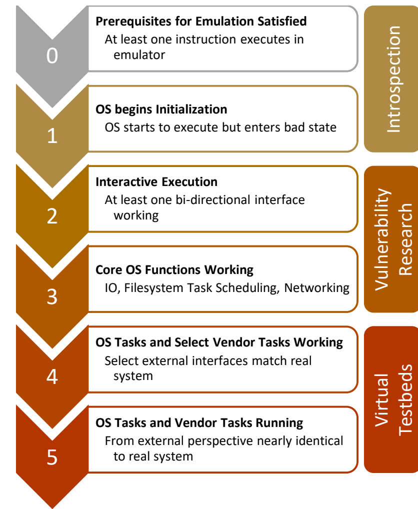

Fig. 1. Re-hosting Utility Scale

B. Emulation Utility Scale

To aid in communicating the utility of an emulated system, we have created a scale to describe the capabilities a re-hosted firmware provides. The level of utility needed from an emulated system depends on the reason for emulating it. The scale is inspired by the levels of autonomous driving [1]. Our scale goes from zero to five, with zero having the least utility and five having the most.

As can be seen in the Figure 1, **Level 0**, means that at least one instruction executes in the emulator, implying that we have the ISA identified correctly and firmware loaded at the correct base address. At this point we can choose any address and start executing from it, but execution will likely not be meaningful. **Level 1** begins when execution starts at the correct entry point and the system starts to properly initialize. At this point we likely have missing hardware requirements that will cause the firmware to not finish initialization. Levels 0 and 1 enable inspecting execution, but is not useful for much more than manual analysis of a few key points.

Level 2 begins when at least one bi-directional interface is working. Examples include reading/writing files, networking, serial ports, or an interactive shell. At **Level 3** all the core OS functions are working including file system, task scheduling, and networking. With a functional interface, levels 2 and 3 become useful for manual and automated vulnerability research like fuzzing.

After completing core OS functionality, device specific applications should be at least partially functional. **Level 4** begins when one or more of the device specific applications are

working and it implies that select applications should behave similar to the physical system. **Level 5** is achieved when all OS and device specific applications are working. At this point, from an external – black box perspective – the emulated system should behave like the real system. It is important to note that Level 5 does not mean it is identical to the real system, as there are likely differences in timing and internal state from the real system. If you want perfectly fidelity you probably need to use the real system. The process of developing and adding our re-hosting layer follow the pattern of pushing the utility of the emulated system up this scale by focusing on the interfaces and software layers needed to move from one level to the next.

C. HALucinator

This work substantially extends HALucinator, and thus we review key parts of its architecture and design principals here. For a more thorough description we refer you to the original paper [8]. HALucinator provides a framework to perform High Level Emulation (HLE) for bare-metal firmware. It does this by identifying hardware abstraction functions in the firmware and replacing them. These functions abstract low-level hardware operations into high-level API's that simplify development.

HALucinator implements replacement functionality using three components in an attempt to maximize code reuse. These components are handlers, peripheral models, and external devices. The handlers interact directly with the emulator's state (i.e., registers and memory) to read/write relevant data to or from the emulated system's memory. Handlers are also the least portable between firmware, as they are implemented specifically per abstraction layer. The handlers often make calls to a peripheral model, where the model implements the behavior of an on-chip or external peripheral, such as a UART, or Ethernet controller in a generic way. This enables reuse of the models in re-hosting multiple firmwares, enabling the handler to be a thin translation layer, mapping the semantics of the firmware's API to a generic model of a peripheral.

To enable the peripheral models to receive input from and send output to a variety of devices, the peripheral models leverage HALucinator's IO Server. The IO Server publishes and receives tagged messages to and from external devices. The external devices are usually scripts that tie host system resources to the emulator. In this work we refer to the set of handlers and models needed to enable re-hosting a class of firmware as a re-hosting support layer (RSL).

The HALucinator paper showed that HLE can be used to re-host firmware in an interactive manner and it also enables automated vulnerability discovery for small bare-metal systems. In this work, we extend the HALucinator platform to enable re-hosting commercial firmware using the VxWork RTOS by developing a re-hosting support layer for VxWorks.

D. VxWorks

VxWorks started as a set of enhancements to VRTX [23]. In the late 1980's, Wind River developed its own kernel to replace VRTX within VxWorks. Over time, VxWorks went from support for 32-bit processors to becoming an RTOS with a networking stack and eventually 64-bit processing in the 2010's. VxWorks supports ARM, Intel and Power architectures, both 32-bit and 64-bit. One of the key features of VxWorks is that the OS kernel is separate from applications, board support packages, middleware, and other packages. This allows for updates of the OS and components, without having to change the applications. These same abstractions help facilitate re-hosting the firmware using HLE.

Our example firmware use VxWorks 5.5 and 6.4, so we focus on their details here. VxWorks 7 brings significant changes and we leave re-hosting it to future work. Figure 2 shows generalization of some key layers in VxWorks relevant to our re-hosting support layer. It starts from the top with applications and moves down through the software stack eventually reaching hardware. Conceptually, a re-hosting support layer needs to replace blocks such that every vertical column has a replacement before reaching hardware. Doing this will enable the re-hosted firmware to emulate the system it implements.

Starting at the top of Figure 2 we have a set of applications that are implemented as tasks in VxWorks. A number of these are provided with VxWorks and others are device specific. Not all the VxWorks applications will be present or executing on every system. Below the application layer, VxWorks provides a POSIX-like API through its IO Subsystem. It provides functions such as *open*, *close*, *read*, *write*, etc. This is a thin layer that looks up the driver that should handle the operation, and forwards the parameters to the appropriate driver functions. Drivers are registered with VxWorks by using calls to *iosDrvInstall* which registers the Create, Remove, Open, Close, Read, Write, and IOCTL functions for the driver and returns a driver. The driver is then passed to *iosDrvAdd* which associates it with a path. These two functions and the drivers associated with them are featured prominently in our re-hosting support layer.

VxWorks provides drivers for file systems (e.g., DosFs) tty devices (serial ports), and network devices. In addition to drivers provided by VxWorks, device specific drivers can also be added. The VxWorks drivers rely on device specific BSP implementations to perform their operations. VxWorks provides its own TCP/IP stack with the ability to register device specific protocols in the networking stack. Connection of these protocol stacks is done through VxWorks' Networking Mux interface. This provides an interface to register device specific drivers conforming to the MUX API. Implementing these functions enables data to be sent and received over the network. One of these types of interfaces is the Extended Network Device (END), which is used for Ethernet devices and is the most applicable to our re-hosting support layer.

III. VXWORKS RE-HOSTING SUPPORT LAYER

With this overview of VxWorks we are now ready to describe our VxWorks re-hosting support layer. We start by laying out its pre-requisites and then proceed through each of the key components in the order that we recommend adding them. This order pushes the utility of the emulated system up the emulation utility scale. Our re-hosting support layer enables initializing VxWorks, executing multiple threads, triggering interrupts, serial communication, a DOS file system, and Ethernet communications.

A. Pre-requisites

Obtain Firmware. Before re-hosting firmware we need to obtain the firmware and determine how to load the firmware into HALucinator. The firmware used in this paper was all obtained by downloading it from their respective manufactures. In all cases, the firmware was a binary file in the downloaded

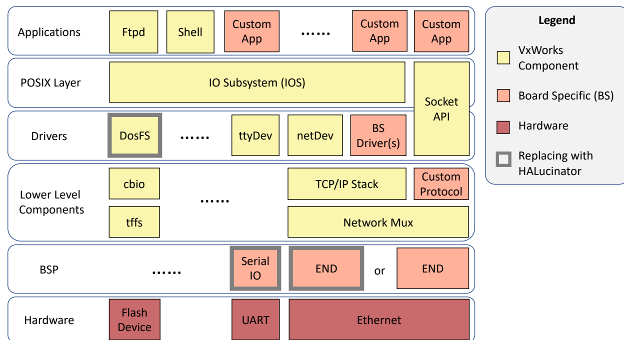

Fig. 2. VxWorks Layer Diagram. This is an example and is generic. Some of the boxes shown here may change, disappear and others may appear depending on the specific device.

package (i.e. the firmware was at most compressed, not encrypted or otherwise packed). In general, obtaining firmware can be a significant challenge [29], [35], [34]. As our goal is advancing the capabilities in re-hosting firmware, we chose devices for which obtaining the firmware was straight forward.

Determine Architecture. After obtaining the firmware we have to determine the processor architecture, load address for the firmware, memory map, and entry point to re-host the firmware. The processor architecture can be obtained by physical examination of the hardware on which the firmware runs, looking for strings in the firmware, or using tools such as binwalk [24] that use a heuristic based analysis to guess the processor architecture by examining the binary and looking for instructions. In our case, a combination of physical examination, strings, and heuristic approaches were used. All of our examples run on ARM micro-controllers that implement the ARM v5T instruction set [16].

Determine Load Address and Memory Map. Getting the correct load address for the firmware is essential for correct execution. If the processor chip is known and datasheets are available, much of the memory map can be obtained from there. Otherwise, we determined the correct loading addresses by loading into Ghidra [19] and examining the correctness of the disassembly after running Ghirda's default auto-analyses. We then used trial and error and examined the use of constant addresses to determine the correct load address. With the firmware loaded into Ghidra at the correct offset, we then use Ghidra to determine the rest of the memory map by looking for references to missing memory. We then add memory to satisfy these missing memories. For the purpose of re-hosting, over-provisioning memory does not impact the execution as it will just not be used. Thus, if it looks like a memory range may be accessed, we add a block of memory to enable the access whether or not it will actually occur. We also leverage dynamic analysis to identify additional regions of memory and iteratively add these as they are discovered.

Symbol Discovery and Function Naming. To effectively replace a layer in a firmware, we need to identify the addresses of the functions. The re-hosting support layer utilizes function names to identify functionality to replace. Labeling functions in firmware can be a challenging problem – LibMatch [8] and other solutions [37], [13], [22], [17] can be referenced for solving this problem. On VxWorks 5 and 6 these techniques are not needed, as they embed a partial symbol table in the binary. Tools such as Ghidra's "VxWorksSymTab_Finder.java" [20] and VxHunter [38] can extract function names from this table, and we leverage them to map symbol names to addresses in HALucinator. This enables us to refer to the functions by their symbolic names in configuration files for HALucinator. This increases both the readability of the configurations files and their portability between different firmware.

Determine Entry Point. Using the symbols, we look for the instruction that should execute first. We do this by looking for callers of the function *usrInit*. Tracing back up the call chain until we can go no further has led to the entry point where VxWorks starts execution. It is important to note that on the real system, a boot loader likely ran before this instruction, so it may be necessary to fix up some memory state for VxWorks to initialize properly. In addition, the firmware may have multiple ways of starting up (e.g., hard power ups, soft-reset, factory reset, etc). This can mean there are multiple ways that usrInit can get executed, each with a different set of assumed events that occurred previously. We generally try to boot into a hard power-up mode, performing a factory reset if possible. This setup will rely on the least amount of configuration being present at startup. How this is done is device specific, but the initialization process will generally indicate what state needs to be set and provide clues about expected values.

B. Logging

At this point the firmware should execute with a Level 1 utility (i.e., it will start initializing but likely fail quickly). One of the things we have found indispensable in pushing the utility to higher levels is to intercept and log error messages. This enables detecting when errors occur, their source, and often get a human readable message. Our re-hosting support layer provides the ability to capture *errnoSet*, and map it to a human readable string using values from [9]. We also capture calls to various device specific error and debug functions that were identified in the process of re-hosting. These are usually human passed readable strings providing clues about the nature of the error.

C. Initialization

VxWorks has a defined startup process that follows a set of steps to initialize memory and call additional bootstrapping code. The specific steps vary between systems, but generally include loading ROM code into memory and setting initial values at set addresses. One example of this data is the boot line, which allows certain parameters, such as network addresses and boot devices, to be set in a string. This string is loaded to a fixed memory address for the system that can be accessed later by bootstrapping code. If this startup code is not available, it is still necessary to initialize memory before jumping to bootstrapping code.

The system starts its kernel bootstrapping by calling a run-once root task, *usrRoot*, which is the first task executed by the kernel. This task is responsible for initializing hardware and spawning additional system tasks, such as serial communications and networking. Since the underlying hardware does not exist in the re-hosted system, we must replace functions which initialize hardware in order to prevent the system from hanging.

Both the memory initialization and kernel bootstrapping will be device specific. However, we have broken the

VxWorks re-hosting support layer into generic components that can be reused depending on the system. For example, the SCADAPack initializes hardware components for serial communications, a DOS file system, and networking. This combination of devices is specific to the SCADAPack, but the initialization routines rely on drivers which can be replaced to provide generic hardware support.

While bringing up a system for the first time, it is often the case that each of these drivers will be added one at a time to ensure they are properly handled. In our case, each component can usually be disabled by simply skipping the initialization routines for that component. Any components that are not essential to the system or analysis can remain disabled. We apply an iterative process of enabling a component and trying to execute to the end of initialization (e.g., the end of *usrRoot*). If a new error is encountered or we get stuck somewhere, we examine the cause and address it by fixing up the offending device-specific code. In practice, this usually requires little more than skipping low-level hardware initialization functions that are waiting on hardware that is not present in the emulator.

For our sample firmwares, we do not execute the bootloader prior to starting VxWorks. Thus, it is necessary to set the bootline. We do this by intercepting the call to *bootStringToStruct* and writing our boot string to the address of the pointer in the first argument of the call. We then allow execution to continue as normal. In this way our bootline gets injected before it is used for initialization.

D. Interrupts

One of the primary reasons to use an RTOS is to enable a processor to perform asychronous tasks. The use of interrupts is essential for enabling this. VxWorks provides its own abstraction for managing interrupts and uses a method called *intConnect* to associate a callback function and priority with each interrupt. A generic interrupt handler is then called, which uses a BSP function to lookup the from the IRQ controller what interrupt occurred and map it to a VxWorks interrupt vector number. This method is usually called *xxxIntLevVecChk* where the xxx is a device specific identifier chosen by the developer. *xxxIntLevVecChk* is passed two parameters by reference; the first points to a level and the second points to the VxWorks vector number passed to *intConnect* for the interrupt that was triggered.

We implement interrupts by adding a simple interrupt controller to QEMU that has an array of memory mapped bytes, which if any are set, triggers an interrupt. We intercept the call *xxxIntLevVecChk* which will get executed to determine what interrupt occurs. We then lookup in our interrupt controller the interrupt source and return the value. VxWorks will then execute the handler for the desired interrupt.

E. Supporting Asynchronous Tasks

VxWorks has two modes for the scheduler to run in: synchronous and asynchronous. Our example firmwares utilizes the asynchronous scheduler that executes in a periodic manner. This is done by configuring a timer to trigger an interrupt at the period with its associated ISR executing the scheduler. We provide this functionality in our re-hosting support layer, by intercepting the execution of *sysClkEnable* and replacing it with a model that starts a timer. On expiration of the timer we trigger the interrupt associated with the system

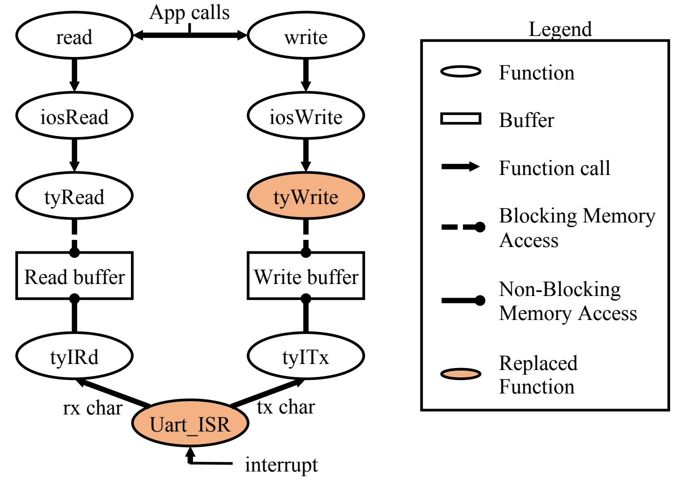

Fig. 3. Shows the read and write operations of serial devices implemented using the VxWorks ttyDev driver, and conceptual replacements made in our re-hosting support layer.

clock. We determine the interrupt used to trigger the clock by statically examining calls to *intConnect* and looking at those that are in the call tree of *sysClkInit*. We also intercept *sysClkDisable* with a handler that stops the timer to prevent the interrupt from occurring when it should not.

F. Serial Communications

Serial communication is used for talking to other devices and for debugging. VxWorks provides a tty driver [31] which provides terminal like behavior to a serial port and enables POSIX file API's to be used for serial port interaction. This driver registers the POSIX layer functions *creat*, *delete*, *open*, *close*, *read*, *write* and *ioctrl* with the IO subsystem. Applications can then open the serial ports like a file and use read/write and ioctl calls to configure and interact with them.

Figure 3 shows the events that occur when a call is made to read or write from an application. Consider a call to *read* on its execution it calls the IO subsystem's read function (*iosRead*). *iosRead* looks up the driver that should handle the read and then calls the driver's read function. For *ttyDev* devices, this is *tyRead*, which attempts to take a semaphore before reading from a receive buffer. The semaphore will be given if there is data to be received or it will cause the task to be removed from execution until data is available and the task can be rescheduled. Writing works similarly, except a semaphore is used to ensure there is room in the write buffer before writing the data.

The ttyDev driver provides functions for the low-level device driver to use for reading (*tyIRd*) and writing (*tyITx*) serial data. The UART's ISR calls these functions to put data into the read buffer and get data from the write buffer. In this way, by defining a few low-level functions, different serial port hardware can easily be integrated into VxWorks.

To implement our re-hosting layer for serial, we replace *tyWrite* with a handler that reads out the data and sends it out HALucinator's IO server. This makes it so data is always sent immediately and removes the need for triggering an interrupt when data is sent. If devices directly use VxWork's ttyDev device, without customization, receiving serial data is done by triggering an interrupt when the data is received for the serial device from the IO server. Then, intercepting the device

specific UART_ISR and calling *tyIRd* with the data. However, different receive models are possible and the SCADAPack uses a receive task to get the data from the serial port. In this case, its receive ISR releases a semaphore that the receive task blocks on. The receive task then reads the data in and calls *tyIRd* (actually called *utyIRd* on SCADAPack). To accommodate this, our serial ISR handler for the SCADAPack modifies the state to indicate to the original handler that data has been received and then allows it to execute. The receive task will then execute *tyIRd*, which we intercept to inject the received data prior to its execution; and then we allow it to execute.

In addition to the handlers supporting reading and writing, handlers are added to monitor ioctl calls. Inserting the serial port layer requires identifying the addresses of *tyWrite*, *tyIRd*, *tyIoctl* – which were in the symbol tables of our firmware samples – and the serial port ISR function. We find the serial port ISR by examining calls to *intConnect*.

We currently intercept the low-level driver functions for polling read/writes, but the same effect can be created by intercepting *tyRead* and waiting until data is received. Because we are using emulated hardware, we can use a virtually unlimited transmit buffer so there is no need to block on transmitting data and our interception *tyWrite* will occur before the low-level write function is called, enabling it to be used as if polling mode is configured.

G. File System

The file system of the SCADAPack, Modicon, and Hughes firmware all utilize the dosFS file system provided by VxWorks. Much like the *ttyDev*, VxWorks installs the file system as a driver using a call to *iosDrvInstall* — registering callback functions that provide read, write, create, delete, open, ioctl and close operations on files. We intercept calls to all these driver functions and map them to the host system in HALucinator. With the file system becoming a directory on the host system.

These functions are not named in the symbol tables in our firmware. For the sake of clarity we will refer to these as *dosFsXXX* where XXX is the operation the function performs (e.g, *dosFsRead*, *dosFsWrite*). In our re-hosting layer the functions *dosFsRead*, *dosFsWrite*, *dosFsClose* are mapped directly to their equivalent function on the host system using python's *os* library. The other functions require mapping VxWorks specific parameters to their equivalent on the host system. *dosFsOpen's* flags need to be mapped to their equivalent opening mode. In addition, VxWorks can create the directories when opening a file by setting a flag. We implement equivalent logic to create directories if this flag is given. Similarly, *dosFsDelete* is used for both files and directories and we handle both depending on the path specified.

The most complex of the functions is *dosFsIoctl*. *dosFsIoctl* takes as one of its parameters a pointer to the device (i.e., dosFS device), the file descriptor, a function code pointer and an untyped value. The function code determines what kind of operation is performed on the file. The 400plus project [21] identified 31 ioctl operations used in VxWorks. We implement the eight different operations we encountered in the SCADAPack firmware. Specifically, fioRead, fioSeek, fioWhere, fioRename, fioReadDir, fioTimeSet, and fioMove. We have also logged attempts to uses of other ioctl operations, enabling us to know if additional operations need to be implemented. Each of these ioctl operations use a different data structure for their value parameter. They follow POSIX defined formats, but offsets for various fields must be carefully determined for each firmware to ensure correct operation.

Adding the handlers for the re-hosting layer requires identifying the addresses of the *dosFsXXX* functions. None of these functions are named in the symbol tables of our firmware samples. However, they are easily identified by examining the call to *iosDrvInstall* in *dosFsLibInit*. Both these functions are in the symbol table of all our firmware samples. In addition, offsets for the data structures used for the ioctl functions may need to be updated. This can significantly increase effort and we are looking for ways to reduce it.

H. Ethernet

The re-hosting support layer supports Ethernet by intercepting the functions used by the VxWorks *muxLib* interface to talk to Extended Network Devices (ie. END in VxWorks documentation) [33]. Specifically, we intercept functions specified in the *END_OBJ* passed to *muxDevLoad* [32] Section 10.1. This structure defines the callback functions used to manage the Ethernet devices. It includes operations to send data, manage multi-cast addresses, load and unload the device, start and stop the device, and handle IOCTLs. We implement handlers for *xSend* which reads out the frame data and sends it using the IO Server. We also implement handlers for *xStart* and *xStop*, which enable and disable receiving of frames and triggering interrupts when frames are received at the IO Server. All other functions currently in the *END_OBJ* are stubbed out to enable monitoring their execution, but execute without modification. These stubs can be removed to improve re-hosting performance.

While the functions to send Ethernet frames are part of the *END_OBJ*, receive functions are not in the *END_OBJ*. VxWorks intends for developer defined code to receive the frame and call *muxReceive* after putting the frame in the appropriate network frame data structure. Further complicating receiving is that to minimize delaying other interrupts, *muxReceive* should not be called in the ISR. Thus, the ISR signals the *netTask* to do the actual reading of the frame and calling of *muxReceive*. Our re-hosting support layer intercepts the Ethernet ISR and makes a call to *netJobAdd* passing it the address of the function used to perform the receive outside the ISR. *netJobAdd* releases the semaphore in the *netTask* is waiting for, enabling it to be scheduled for execution. Upon execution, *netTask* calls the *ENDReceive* function. We intercept this function and replace its logic with functionality that: (1) reads the frame from the Ethernet model, (2) allocates the network data structure by making a call to *netTupleCreate*, (3) copies the Ethernet frame into the structure, and finally (4) calls *muxReceive* with the structure.

Locating the functions needed to add the Ethernet in the re-hosting support layer, requires manual reverse engineering as the functions intercepted are device specific. We locate the *END_OBJ* functions by statically and dynamically examining calls to *muxDevLoad*. *muxDevLoad* is a VxWorks function and was found in the symbol tables of all our sample firmware. The receive functions addresses were found by examining calls to *intConnect*. This gives the ISR for receiving the frames, and we then examine the ISR's for calls to *netJobAdd* to find the function used by the netTask to receive the frame. Using our Ethernet re-hosting layer we are currently able to send and receive an Ethernet frame and get a successful

TABLE I.
SYMBOLS RECOVERED USING GHIDRA VXWORKSSYMTAB_FINDER

Firmware	Functions Recovered	Total	%	Symbols Recovered	Total	%
ScadaPack	4,904	6,440	76%	6,111	15,386	40%
Modicon	8,057	15,324	53%	16,914	23,004	74%
Hughes	19,164	22,221	86%	22,274	42,503	52%

response to an ARP query. Work is ongoing to optimize the receiving of frames to enable more complex interaction.

IV. FIRMWARE ANALYSES

To analyze the effectiveness of our re-hosting, we first examine the success of extracting the symbol table from our example firmware. We then demonstrate the utility of our re-hosted firmware by performing a series of analyses of interest to a firmware analyst. These include recovering the file systems, enumerating the tasks created, inspecting execution traces through custom tools we have built for Ghidra, and enabling the VxWorks shell when not initialized.

These analyses and our re-hosting capabilities are a work in progress. Thus, we focus primarily on the SCADAPack 350 as it is the firmware we have spent the most effort re-hosting. Typically, we develop for it and then port our solutions to the other devices. Currently, we are able to re-host the SCADAPack's firmware well enough to send and receive serial data and send and receive Ethernet frames. Porting to the Modicon and Hughes are a work in progress and we are still working to complete their initialization. We report our findings to date for all devices.

A. Symbol Recovery

Recovering of symbols is essential for the portability of our re-hosting support layer, as it enables the rapid adaptation of the layer to new firmware. Our firmware samples use VxWorks 5.5 and VxWorks 6.4 which embed a symbol table in the binary. We use Ghidra 9.2 [19] to load the binaries into memory using the offsets and memory layout determined manually. We then use the *VxWorksSymTab_Finder.java* script that ships with Ghidra to read the symbol table and populate it into a Ghidra project, after which Ghidra's default auto analyses are run to identify functions and other references. Table I shows the number of function names recovered, the total number of functions identified by Ghidra's analyses, the total number of symbols read, and the total number of symbols Ghidra reported. In the worst case, 53% of the functions identified by Ghidra have names in the symbol table. In the best case 86% of the functions are recovered.

B. Re-hosting the firmware

We re-host the three firmware samples and let them execute as far as they can. Table II shows the functions we intercept and replace with our firmware re-hosting layer for each firmware. For symbols present in the symbol table a check mark is also given. Empty columns mean the handler did not execute on that device. Starting with the SCADAPack, you can see that it uses the Ethernet, DosFS, and ttyDev drivers. The majority (13) of it's handlers are to handle device specific initialization. Of these, all but *at25ReadNBytes*, *eepromReadNBytes*, and *detectBootType* skip the function or return a constant value. *at25ReadNBytes* and *eepromReadNBytes* are both device specific functions that execute the same handler. The handler returns values from a file that was created by reading the contents of the eeprom on the physical system. *detectBootType* writes a value to a global variable to indicate that the boot mode is a factory reset. The value was determined by examining callers to *detectBootType*. These handlers highlight some of the challenges with properly initializing the system that do not port well from device to device. Without access to the physical device, we would have had to manually reverse engineer callers to *at25ReadNBytes* and *eepromReadNBytes* to determine the valid values to return.

TABLE II. FUNCTIONS INTERCEPTED DURING INITIALIZATION.

Grouping	Handler	SP	Exe	Mod.	Exe	H.	Exe
DosFS	dosFsCreate		2				
DosFS	dosFsDelete		3				
DosFS	dosFsIoctl		9				
DosFS	dosFsOpen		9				
DosFS	dosFsClose		11				
DosFS	dosFsRead		11				
DosFS	dosFsWrite		6				
DosFS	iosDevAdd	✓	6	✓	10	✓	8
Error	errnoSet	✓		✓	16	✓	170
Ethernet	eth_Ioctl		26				
Ethernet	eth_Send		1				
Ethernet	eth_Start		2				
H. Init	0xa0052280						1
H. Init	0xa042c1d8						192
H. Init	flPollTask	✓				✓	1
H. Init	FUN_a0010830						2
H. Init	FUN_a049242c						1
H. Init	FUN_a04ac780						1
H. Init	FUN_a04ac804						768
H. Init	inmFlashErase					✓	2
H. Init	sysSpin					✓	16
H. Init	usrMmuInit					✓	1
Init	bootStringToStruct	✓	1	✓		✓	1
Interrupts	xxxIntLvlVecChk	✓	57	✓		✓	203
Logging	intConnect	✓		✓	6	✓	40
Logging	intDisable	✓		✓	2	✓	6
Logging	intEnable	✓	22	✓	11	✓	13
Logging	intExit	✓	56	✓		✓	
Logging	taskSpawn	✓	24	✓	13	✓	13
Mod.	SendToDisplay			✓	1		
Mod. Init	CntSyst1MicroDeltaTime			✓			1
Mod. Init	FUN_2022bd5c				2		
Mod. Init	kl_NandFlashDrvInit			✓	1		
Mod. Init	taskDelay			✓	8		
Mod. Init	usrToolsInit			✓	1		
SP Init	FUN_020a6c10		1				
SP Init	at25Close	✓	2				
SP Init	at25Open	✓	2				
SP Init	at25ReadNBytes	✓	2				
SP Init	cbioDevVerify	✓	1				
SP Init	defaultSerialInit	✓	1				
SP Init	detectBootType	✓	1				
SP Init	eepromReadNBytes	✓	16				
SP Init	flCall	✓	3				
SP Init	getclock	✓	2				
SP Init	readBattery	✓	1				
SP Init	usbPhciInit	✓	1				
SP Init	writeSH3000Register	✓	5				
SysClk	sysClkEnable	✓	1	✓		✓	1
ttyDev	tyIoctl	✓	5	✓		✓	
ttyDev	tyRead	✓	1	✓		✓	382
ttyDev	tyWrite	✓		✓	33	✓	1017

*Check-mark means the symbol is recovered with default symbol finding script for the given device.

TABLE III. COMMON NAME TASKS CREATED

Task Name	Scadapack	Modicon	Hughes
tExcTask	✓		✓
tLogTask	✓	✓	✓
tNbioLog	✓	✓	
tTffsPTask	✓		✓
tNetTask	✓	✓	✓
tFtpdTask	✓		✓

TABLE IV. DEVICE SPECIFIC TASKS

Scadapack		Modicon	Hughes
t0	com1Tx	tJobTask	tEthSend
BULK_CLASS	masterReq	tErfTask	USB_RXTX
BULK_CLASS_IRP	mrTx1	tDhcpcStateTask	tTelnetd
tBulkClnt	com2Tx	tDhcpcReadTask	tBridgeAger
tRxTask0	mrTx2	tTftpd	tDhcpsTask
tRxTask1	com3Tx	tFtp6d	***tShell***
tRxTask2	mrTx3	tStartSnmpd	tPicDriver
fileWatch	mTcpServ	MidRangePPP_D_1	SYSLOG
com1Rest	mruServ	MidRangePPP_C_1	
com2Rest	mauServ	EnableUSB	
com3Rest	mastIpSta	tXbdService	
Startup	bkgrndIo	t920_PollTask	
udpConfig	softWatch		
ioCtrl	socketWatch		
ioRecv	HALThread		

Of the three devices, the SCADAPack gets furthest through its execution. It completes execution of its initialization and is waiting to be configured. We boot it in a factory reset mode and so it has no control logic to execute. In addition, the device can speak a variety of control protocols over its serial ports. This means that in actual use they are likely connected to other devices and it is essential that non-protocol data–like debug strings and boot info–not be written to them. We can see this by the lack of calls to *tyWrite*. We are able to get it to receive and process both serial and Ethernet data, but this functionality was not exercised in the capture used to generate Table II.

The our re-hosting support layer for the Modicon 340 is the least mature of our samples. It currently gets stuck in the last function call of its *usrAppInit*–the last high level function of it's initialization. It is waiting on data from a SD Card that we have not yet implemented. From the handlers intercepted you can see that it is writing to a ttyDev. To get this firmware to the same point as the SCADAPack, we need to get the interrupt vector table to initialize and the start the system clock. At that point we can stop intercepting taskDelay–which we currently skip to prevent hanging forever. This allows us to get further in initialization to determine what resources the system uses, but prevents other tasks from being scheduled. We also need to determine how the system initializes its Ethernet interface as it does not appear to use the boot line for this purpose.

The Hughes 9201 BGAN completes its initialization, but does so with a large number of errors. It calls errnoSet 170 times and many of these are related to the creation of network devices and file systems. We are working to continue fixing up it's initialization so that these components will work. Even with these limitations we can learn some interesting things about the Hughes device. For example, it uses a ttyDev for printing out debugging information and runs the VxWorks shell. Figure 4 is a screen capture of the shell initialization. We can also see that number of symbols recovered is only one less than the firmware finds (e.g. 22,274 vs 22,275).

C. Task Analysis

Analyzing the tasks that firmware starts can tell an analyst a lot about what the system does and how it does it. To

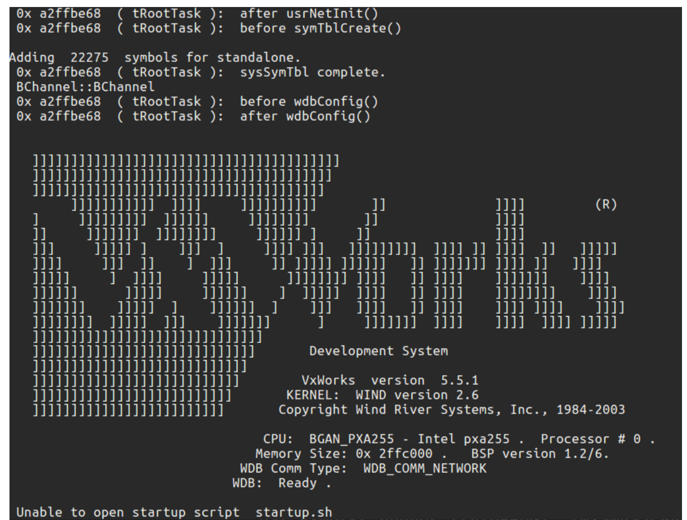

Fig. 4. Output from Hughes 9201 ttyDev during initialization

support this type of analysis we add a handler that logs the calls to the function used to create new tasks *taskSpawn*. We log all the parameters which gives us the task's: name, stack size, priority, entry point, flags, and parameters passed to the entry point. The common tasks created through their initialization process are shown in Table III and device specific tasks are shown in Table IV. From this you can see that all three run a logging task (*tLogTask*) and a networking task (*tNetTask*). The SCADAPack and Hughes device also both run VxWork's FTP server (*tFtpdTask*). The modicon device also appears to runs a FTP server *tFtp6d* and a Trivial FTP server *tTftpd* but these are specific the modicon device.

Looking at device specific tasks we can see that the SCADAPack starts a number of tasks for each serial port (tRxTaskN, comNRest, mrTxN, comNTx). The fact each serial port is managed by four tasks indicates complex and critical behavior occurring on the serial ports. This makes them of interest for further security analysis. Looking at the Modicon and Hughes device we also find a number of network facing services (e.g., DHCP, SNMP, Telnet) that could be of interest to security analysts.

As previously mentioned, the Hughes device executes a VxWorks shell over a serial port. We have been able to interact with the shell using re-hosting and use it to execute commands, inspect the system, and execute scripts. However, because of all the errors during initialization we cannot yet say whether this interface would be unprotected on physical systems. It is possible that later execution, or a missing part of initialization disables it or sets a password on it. In addition, it possible that physical protections like not routing the serial pins to pads on the PCB are used to protect the interface.

D. Discovering and Enabling the VxWorks Shell

VxWorks provides an interactive shell designed for interfacing with the operating system. The shell serves two purposes: as a command interpreter and for prototyping and debugging. The command interpreter is similar to a UNIX shell where commands are issued to exercise system functionality. The prototyping and debugging features allow for low-level introspection and control into tasks running on VxWorks. This allows the user to set breakpoints in code, call user routines,

examine and modify memory, and catch errors on the system.

The shell must be included in the application by the developer. This is done by calling *shellInit* in the bootstrapping task, *usrRoot*. The shell would likely be included during system development since it is the primary mechanism for debugging applications. Each of the three devices we inspected contained the shell symbols, but not all started the shell task at startup. Since we control execution of the re-hosted firmware, we can enable the shell by inserting our own call to the *shellInit* function at startup.

Inspect Running Tasks. Going beyond the knowledge of what tasks have been started. The shell allows tasks to be listed and inspected dynamically. Showing what tasks are *running* at any given time. Additionally, the shell can show information about a task, such as the location of its stack, and spawn or suspend tasks. Comparing running tasks to started tasks can indicate where re-hosting fidelity needs to be improved.

Modify Filesystem. The shell can be used to traverse the device filesystem and modify files using a file editor. Modifying scripts on the re-hosted device filesystem can persistently modify behavior of the system. For example, some devices rely on scripts at startup for initialization.

Read/Write Memory. Memory can be read and written using shell commands. This can be used to leak information or modify the memory of a running process.

Run User Routines. The shell has a table of symbols which can be invoked from the command line. This mechanism can be used to run arbitrary user functions on the device.

E. Ghidra Integration

In many cases during the re-hosting process something goes wrong. Determining where this occurred and understanding it accelerates the re-hosting process. To support this we have developed a tool, called HQ-Tracer, which parses the execution log from HALucinator/QEMU and loads it into Ghidra. This tool colors all instructions executed in the trace and highlights the current instruction in the trace. It also enables interactively stepping through the traces in Ghidra using a command-like interface similar to GDB. These commands include stepping forward and backward X number of instructions (default of 1 at a time), continue until you exit the given function, and it allows to search for a specific address forward/backward. If you only have a single thread/task, there is a stack tracker that attempts to keep a call stack to quickly see the function call path to the current instruction, though this feature is disabled by default and can be toggled on. In addition, the trace enables executing python code as a command which enables performing customized filtering and stepping through the trace. During our debugging and discovery phases of re-hosting, we often would not understand why some things were not working. Loading the trace into Ghidra enabled us to quickly determine the location of the error and determine work arounds.

V. Approach For Re-hosting Other RTOSes

In re-hosting VxWorks we discovered multiple places where HALucinator needed extensions, and the approach used to re-host firmware in the original HALucinator paper needed refinement. Based on this experience we propose an approach to use as a starting point when re-hosting an RTOS. In this approach, we assume you can get the firmware loaded into HALucinator, identify the entry point and begin execution (i.e., you have reached Emulation Utility Level 1). We also assume that key functions of the RTOS have been identified. This can be done using symbols, function labeling techniques, or manually. If it needs to be done manually, then this approach can serve as a template to help identify what these function need to be identified. This process is primarily derived from the approach we took, but incorporates ideas and suggestions that we wish we had known starting out. We believe these ideas can shorten the process of building a re-hosting support layer for other operating systems.

(1) Instrument error handling and print statements. Once the execution in the HALucinator can begin, we recommend intercepting and logging calls to error handling functions first. This provides a high level way to track what the firmware is doing and determine where execution is going wrong. If these can be mapped back to human readable strings this is particularly useful, as it provides hints of how to fix the issue or even determine if the issue needs resolved. For example, if the error is for an interface you don't intend to use in the emulated system, then you may not need to fix it. In addition to error handling code instrumenting, print statements (e.g. *printf*, etc) provide indication of what is being executed and if things are going as expected.

(2) Understand and execute RTOS initialization to completion. With error handling and print statements instrumented, the next step is to get the OS initialization process to complete. This will require understanding the boot process for the OS and determining when initialization is complete, i.e., the system switches to its steady state operations. For VxWorks, this is generally the completion of execution of *usrRoot*. The initial goal is not to get everything to initialize correctly, but rather to get execution to reach the end of initialization. This lets you know what needs to be initialized and then to selectively work on individual components as needed.

We did this by stubbing out functions that prevented reaching the end of initialization. This includes hardware initialization functions that polled non-existent IO, task delay functions, clock reading functions, and anything else that prevents reaching the end of initialization. We replace these functions with stubs that indicate they executed successfully (e.g., returning zero) or just skipped their execution. For functions reading clock values we return an increasing value on each call.

The goal at this stage is to understand what needs to be initialized. To this end, we delay starting asynchronous context switching as long as possible. This simplifies debugging and execution tracing. If initialization is dependent on another task running and providing a resource, we often skip the execution of this function during the initial re-hosting stage. In this way, we can rapidly identify what needs to be initialized and determine where we will spend our effort in re-hosting.

(3) Identify tasks creation. With the system running through its initialization process you can now identify the tasks that it creates (or at least attempts to), their priorities and usually a human readable name. We do this by intercepting calls to *taskSpawn* and logging the tasks to a file, then allow execution to continue unmodified. We also instrument the task switching code to determine when a task starts execution.

(4) Identify interrupt handlers and driver registration.

With the system running through its initialization process, you can now dynamically identify interrupt handlers, and driver registration. Replacement of these drivers will enable providing support for the hardware missing in the emulator. The drivers and their initialization functions are likely the source of most the errors encountered during initialization. We identify these functions by intercepting the driver registration functions *iosDrvInstall* and *iosDrvAdd*. Intercepting these functions will tell you what kind of devices the system is using and the functions used to interact with the system.

On VxWorks, a number of these drivers are software only components provided by VxWorks and they use another set of lower level callbacks to interact with the hardware. We found it much easier to intercept these lower level callbacks rather than the higher level set. For example, we intercept the utyLib functions (e.g., *tyWrite)* for the serial port. This allows the read and write functions provided by the tty driver to handle the semaphores and interprocess communication needed for multiple tasks to execute on the system in the firmware, rather than having to have our handlers manage these semaphores. Networking is much the same – a high level network driver is added that provides the POSIX file api, but VxWorks intends for network functionality to leverage the MUX interface.

Once you have identified the drivers, we suggest you work to bring them up one at a time, following the order the initialization function tries to initialize them, with one exception. We suggest delaying starting the scheduler for as long as possible, as debugging multiple tasks is significantly harder than a single task. However, we found at some point the initialization process starts other tasks and expects them to execute prior to its completion. In these cases, complete as much of the initialization as you can before starting the scheduler, then enable it. For each driver, work to understand their initialization process and how to determine that it is performed correctly. Then for each interface you care about, ensure that each driver initializes as completely as possible.

(5) Starting Non-cooperative Context Switching. When either initialization is completing without errors or initialization is dependent on having other tasks run, it is time to start non-cooperative context switching. This is done by starting a timer that triggers the system tick ISR periodically. We recommend to initially start slow, and then increase rate as needed for performance. At this point the operating systems should be running along with tasks that are not dependent on unsupported devices.

VI. Related Work

The first large scale academic works that enabled system-emulation and firmware re-hosting were Firmadyne [7] and work by Costin that is very similar [10], [11]. The ideas behind these works include extracting the file system from a Linux firmware, and re-hosting using their own Linux kernel using QEMU. Their tools work only for Linux firmware that can natively use chroot inside of QEMU. After re-hosting firmware, they perform static and dynamic analysis to report vulnerabilities.

The group that created Avatar [36] also created Avatar^2 [18], which was completely re-designed from the original Avatar implementation. Avatar^2 is a dynamic multi-target orchestration and instrumentation framework that allows various other tools (angr [28], QEMU [6], GDB, HALucinator [8], etc.) to integrate and communicate state between them during dynamic execution. It can also be used to perform hardware in the loop emulation of systems.

Using the drop-in fuzzer AFL [5], P^{2}IM [12] provides inputs to a modified QEMU emulator for any peripheral or hardware IO. Their approach is unique and different from existing emulation approaches, as it does not use hardware, traces, or detailed knowledge of the peripheral/hardware IO, rather they are relying on the fuzzer to provide all interactions. By using the drop-in fuzzer, it enables simple peripherals to be emulated, but its ability to enable complex data and stateful hardware is still either lacking or unknown. It also does not allow interactive system of system emulation as all IO is randomly generated. Leveraging machine learning, Pretender [14] records hardware interactions and all accesses to memory mapped input and output regions. These recordings are used to train a machine model that during re-hosting provides inputs for missing peripherals. Both of these approaches do not allow external inputs to peripherals making them unsuitable for system of system modeling or use cases beyond vulnerability analysis.

In "Challenges in Firmware Re-Hosting, Emulation, and Analysis", Wright et al. provide a comprehensive overview of the challenges faced during system emulation, firmware re-hosting, and analysis while providing classification and comparison techniques on five different axes [34]. Using the fidelity classification presented in that review, we note that HALucinator and Avatar^2 have higher fidelity than P^{2}IM and Pretender. We refer the reader to this paper for a more comprehensive review of the challenges faced during general system emulation and firmware re-hosting along with a wider evaluation of tools, as our focus is more in depth and focused on embedded devices that use VxWorks.

VII. Conclusion

With the re-hosting support layer we have implemented and deployed for a SCADAPack 350, Modicon 340 and Hughes 9201 BGAN, the time for porting the layer to another device using VxWorks can be significantly reduced. Re-hosting these devices can be used for vulnerability discovery or other general system analysis. We have thoroughly explained our re-hosting support layer, functions and tasks intercepted in VxWorks, and we have given a retrospective approach for re-hosting other RTOSes. We are continuing to expand our capabilities in this area with more devices in the works for re-hosting using our expanded HALucinator handlers and techniques.

Acknowledgment

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] "The 6 levels of vehicle autonomy explained." [Online]. Available: https://www.synopsys.com/automotive/autonomous-driving-levels.html

[2] "Command & data-handling systems." [Online]. Available: https://mars.nasa.gov/mro/mission/spacecraft/parts/command/

[3] "Customer success: Varian medical systems." [Online]. Available: https://www.windriver.com/customers/customer-success/medical/varian/

[4] "CVE-2019-12257." Available from MITRE, CVE-ID CVE-2019-12257., Dec. 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12257

[5] AFL-Fuzz, "afl-fuzz." [Online]. Available: https://github.com/google/AFL

[6] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[7] 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: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/towards-automated-dynamic-analysis-linux-based-embedded-firmware.pdf

[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] C. Co, "Vxworks error codes," Dec 2014. [Online]. Available: http://blog.lovecoco.net/168

[10] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *23rd USENIX Security Symposium*. San Diego, CA: USENIX Association, Aug 2014, pp. 95–110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin

[11] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2016, pp. 437–448. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897900

[12] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling (extended version)," *ArXiv*, vol. abs/1909.06472, 2019.

[13] firminsight, "firminsight." [Online]. Available: https://github.com/ilovepp/firminsight

[14] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel *et al.*, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 2020.

[15] J. Laukkonen, "Will bmw's infotainment solution idrive you up the wall?" Feb 2020. [Online]. Available: https://www.lifewire.com/examining-the-bmw-idrive-interface-534742

[16] A. Limited, *ARM Architecture Reference Manual*, ARM. [Online]. Available: https://developer.arm.com/documentation/ddi0100/latest/

[17] H. Mohanan, P. Bendapudi, A. Kumarasubramanian, R. Jalan, and R. Venkatesan, "Function matching in binaries," Apr 2012, uS Patent 8,166,466.

[18] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar^2: A multi-target orchestration platform," in *Workshop on Binary Analysis Research, colocated with Network and Distributed Systems Security Symposium, San Diego, USA*, San Diego, UNITED STATES, Feb 2018. [Online]. Available: http://www.eurecom.fr/publication/5437

[19] NSA, "Ghidra." [Online]. Available: https://ghidra-sre.org/

[20] NSA, "Vxworkssymtab_finder.java." [Online]. Available: https://github.com/NationalSecurityAgency/ghidra/blob/fe8d863c47f79d904c10c2c49d16ea4c1b674020/Ghidra/Features/GnuDemangler/ghidra_scripts/VxWorksSymTab_Finder.java

[21] E. Perez, "400plus/iolib.h." [Online]. Available: https://github.com/400plus/400plus/blob/master/vxworks/ioLib.h

[22] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in cots binaries," in *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2017, pp. 201–212.

[23] J. F. Ready, "Vrtx: A real-time operating system for embedded microprocessor applications," *IEEE Micro*, vol. 6, no. 4, pp. 8–17, 1986.

[24] ReFirm Labs, "binwalk." [Online]. Available: https://github.com/ReFirmLabs/binwalk

[25] W. River, "Functional safety." [Online]. Available: https://www.windriver.com/functionalsafety

[26] ——, "Security vulnerability response information: Tcp/ip network stack (ipnet, urgent/11)." [Online]. Available: https://www.windriver.com/security/announcements/tcp-ip-network-stack-ipnet-urgent11/

[27] S. Shah, "The arm-x firmware emulation framework." [Online]. Available: https://github.com/therealsaumil/armx

[28] 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 *IEEE Symposium on Security and Privacy*, 2016.

[29] S. Vasile, D. Oswald, and T. Chothia, "Breaking all the things—a systematic survey of firmware extraction techniques for iot devices," in *Smart Card Research and Advanced Applications*, B. Bilgin and J.-B. Fischer, Eds. Cham: Springer International Publishing, 2019, pp. 171–185.

[30] A. Volosincu, "Vxworks: Past and future," Jul 2018. [Online]. Available: https://blogs.windriver.com/wind_river_blog/2018/07/vxworks-past-and-future/

[31] VxWorks, "Vxworks reference manual : Libraries." [Online]. Available: https://www.ee.ryerson.ca/~courses/ee8205/Data-Sheets/Tornado-VxWorks/vxworks/ref/libIndex.html

[32] *VxWorks Network Programmer's Guide 5.5*, Wind River, 500 Wind River Way, Alameda, CA, 94501, 8 2002. [Online]. Available: http://www.ing.iac.es/~docs/external/vxworks.old/Network-Guide-5.5.pdf

[33] *VxWorks Network Protocol Toolkit User's Guide*, Wind River, 500 Wind River Way, Alameda, CA, 94501, 8 2002.

[34] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys*, 2020.

[35] S. J. Yang, J. H. Choi, K. B. Kim, and T. Chang, "New acquisition method based on firmware update protocols for android smartphones," *Digital Investigation*, vol. Volume 14, pp. S68 – S76, 2015, the Proceedings of the Fifteenth Annual DFRWS Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287615000535

[36] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares," in *Network and Distributed Systems Security Symposium*, Feb 2014.

[37] R. Zhu, Y.-a. Tan, Q. Zhang, Y. Li, and J. Zheng, "Determining image base of firmware for arm devices by matching literal pools," *Digital Investigation*, vol. Volume 16, pp. 19 – 28, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287616000037

[38] W. Zhu, R. Liu, J. Wang, and Y. Zhou, "Vxhunter: A tool set for vxworks based embedded device analyses," in *black hat ASIA 2019 Workshop*. blackhat.com, March 2019. [Online]. Available: https://www.blackhat.com/asia-19/arsenal/schedule/index.html#vxhunter-a-tool-set-for-vxworks-based-embedded-device-analyses-14165