EMULATION FOR MULTIPLE INSTRUCTION SET ARCHITECTURES

by

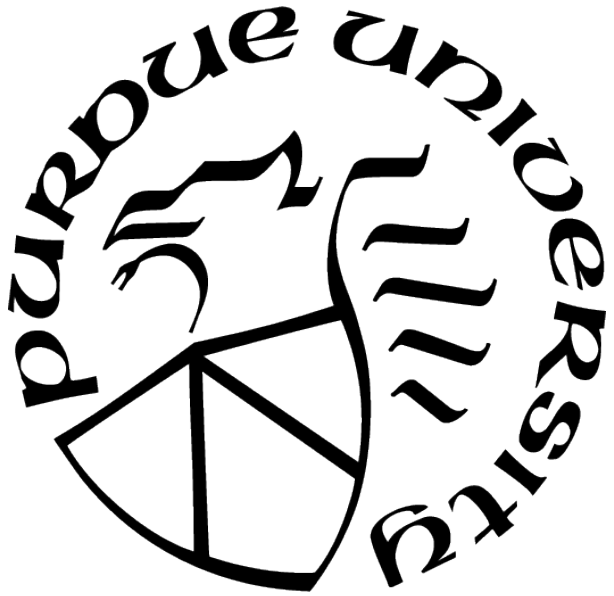
Christopher Wright

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Milind Kulkarni, Chair

School of Electrical and Computer Engineering

Dr. Samuel Midkiff

School of Electrical and Computer Engineering

Dr. Vijay Raghunathan

School of Electrical and Computer Engineering

Dr. Abraham Clements

Sandia National Laboratories

Approved by:

Dr. Dimitrios Peroulis

Dedicated to Jennifer, Autumn, Quinton, and Roman.

ACKNOWLEDGMENTS

Thank you to my wife for her support and help through years of me figuring out my passion. We joke that she should be in my gown with me at graduation, and she really should have an honorary doctorate to go along with mine because of all the extra work she had to do to enable me to do this research. Her encouragement kept me going through the rough patches of research and life. To my children, thank you for your patience and love, you are the joys of my life.

A special thank you to my Ph.D. chair Milind Kulkarni, he has been inspirational, understanding, and kind throughout my years of research. He was always available, helpful, and was a big reason I ended up at Purdue University. He pushed me and encouraged me to be better at research and is someone I try to emulate when teaching. His mentorship and example helped me be the person I am today.

I also wish to acknowledge and thank my Ph.D. committee for their patience and guidance. The courses taken from both Dr. Raghunathan and Dr. Midkiff have helped in creating tools and understanding background material at a higher level for my research. Their courses were top notch, fun, entertaining, and helped shape my research interests. Their questions and insights led me to refine my focus and research into a broader applicability for my tools. Dr. Clements has been a wonderful friend and mentor. He and his family have been there for my family throughout this entire Ph.D. ordeal, and his friendship means a great deal to me.

Thanks to Kanak, Nour, Krish, Laith, Nikhil, Jad, Jianqiao, Charitha, and the rest of the PLCL group for interesting technical conversations and group chats. Finally, thanks to Sandia National Laboratories for their willingness to collaborate and make this research possible.

This research was partially funded NSF grants CCF-1337158, CCF-1725672, CCF-1908504. This research was partially funded and performed in collaboration with Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Secu-

rity 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. SAND2021-4148 T.

PREFACE

My research started with writing a compiler for Bioinformatics and Genomics, which led to research writing new assembly algorithms and parallelization of existing algorithms to scale. I had a background in static and dynamic analysis, programming languages, compilers and embedded systems. My work with Sandia National Laboratories integrated my background into the vulnerability discovery and emulation space, and I was intrigued with the relatively new area of Genomic Security. With my background in Bioinformatics and analysis, it seemed to be the perfect fit. Originally this thesis was to be more focused on the Genomic Systems side of things, but with the COVID-19 pandemic, the availability of various firmwares and support from industry partners was reduced so that was not feasible. During my Preliminary examination the feedback from committee members was that my thesis should stay broader, as the work I proposed was beneficial to the emulation field in general, not just genomic security. This led to the motivation of the work and research I have done in emulation, firmware re-hosting and binary analysis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ARM	Advanced RISC Machine
BMES	Bare Metal Embedded System
CFG	Control Flow Graph
DMA	Direct Memory Accesses
ELF	Executable and Linkable Format
FPGA	Field Programmable Gate Array
GDB	The GNU Project Debugger
GPES	General Purpose Embedded System
HAL	Hardware Abstraction Library
HITL	Hardware in the Loop
HLE	High Level Emulation
HMI	Human Machine Interface
ISA	Instruction Set Architecture
MMU	Memory Management Unit
NVM	Non-Volatile Memory
NVRAM	Non-Volatile Random Access Memory
PLC	Programmable Logic Controller
PMEM	Persistent Memory
RAM	Random Access Memory
RE	Reverse Engineering
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RSL	Re-hosting Support Layer
RTU	Remote Terminal Unit
SPES	Special Purpose Embedded System

ABSTRACT

System emulation and firmware re-hosting are popular techniques to answer various security and performance related questions, such as, does a firmware contain security vulnerabilities or meet timing requirements when run on a specific hardware platform. While this motivation for emulation and binary analysis has previously been explored and reported, starting to work or research in the field is difficult. Further, doing the actual firmware re-hosting for various Instruction Set Architectures(ISA) is usually time consuming and difficult, and at times may seem impossible. To this end, I provide a comprehensive guide for the practitioner or system emulation researcher, along with various tools that work for a large number of ISAs, reducing the challenges of getting re-hosting working or porting previous work for new architectures. I layout the common challenges faced during firmware re-hosting and explain successive steps and survey common tools to overcome these challenges. I provide emulation classification techniques on five different axes, including emulator methods, system type, fidelity, emulator purpose, and control. These classifications and comparison criteria enable the practitioner to determine the appropriate tool for emulation. I use these classifications to categorize popular works in the field and present 28 common challenges faced when creating, emulating and analyzing a system, from obtaining firmware to post emulation analysis. I then introduce a HALucinator [1]/QEMU [2] tracer tool named HQ-Tracer, a binary function matching tool PMatch, and GHALdra, an emulator that works for more than 30 different ISAs and enables High Level Emulation.

1. INTRODUCTION

1.1 Motivation

The number of connected devices, from appliances to smart home and personal gadgets, has increased dramatically with the explosion of Internet of Things (IoT). Along with these everyday gadgets, large scale infrastructures such as the electric grid, cellular networks, and other large control systems have become smarter, digital, and interconnected [3]–[7]. Understanding how these systems work and discovering vulnerabilities in their firmware is an important and growing topic in academic and industrial research, with large companies paying millions of dollars for bugs found in their devices [8]–[13]. The increased effort to protect connected devices has come from an increased awareness of their vulnerability and the attacks targeting them [14]–[22]. Many of these systems have access to sensitive financial data, personal information, or control critical processes. Malicious actors are exploiting vulnerabilities in these systems to cause harm to businesses, individuals, and critical systems such as electrical grids and cellular infrastructure. Some countries have even started to ban hardware made by certain companies because they fear the access these companies will give to their home countries [23].

There are many approaches to protecting and fixing vulnerabilities in systems. Code analysis at the source level is a popular option, including taint analysis, numerical analysis, overflow analysis, binary hardening, obfuscation, etc. Other techniques include randomizing inputs and sending them to the actual system. If this is done on real hardware it can 'brick' the hardware, and in many cases we have no idea why. Emulation has been a growing area of interest for dynamic analysis because of the increasing compute capacity of common desktop and laptop computers. Emulation is popular for architectures that are supported, and for these architectures, emulation can help with Reverse Engineering (RE) certain firmware of interest.

If you have a device that you want to determine is secure or has vulnerabilities, often times you just have the device or maybe only the name of the device. To do an analysis you have to first get the firmware for the device, then determine the architecture, the base address, entry point, encryption, obfuscation techniques and more, all before you can even start to

emulate. Then once you get to a point where you can emulate, you may not understand what is going on because of hardware interactions that fail. It may be tempting to just give up and try to analyze a system where you have source code. Because the binary world is so detailed yet unknown, there is an openness and wildness that is intriguing.

1.2 Thesis Statement

This thesis provides a comprehensive systematization of knowledge in the area of system emulation and firmware re-hosting, providing classification techniques and novel static and dynamic analysis tools and techniques that enable system emulation to re-host firmware with the least amount of effort for the greatest number of instruction set architectures.

To this end, I show that it is possible to reverse engineer and emulate systems to re-host firmware for a wide variety of architectures, even if source code is not available. I first provide a systematization of knowledge review with comprehensive classification techniques, and classify the most popular available tools used for firmware re-hosting. I then discuss the tools and software I have created to aid in Pre-Emulation, Emulation and Post-Emulation challenges. Finally I show integration with the High Level Emulation approach presented in HALucinator [1]. I show that the work done in collaboration with Sandia National Laboratories where we explained and demonstrated how to extend HALucinator by creating a Re-hosting Support Layer to work for the VxWorks RTOS [24] works for GHALdra, a Ghidra HALucinator Emulator. I now introduce the different sections of contributions in high level detail and in subsequent chapters these are discussed and explained in depth.

1.3 Systemization of Knowledge

The systematization of knowledge and classifications and techniques used on current tools is encompassed in Chapter 2, Chapter 3, and Chapter 4. I present the evolution of emulation along with emulator bases in Chapter 2, before introducing the most relevant works in the area today. I introduce comparison metrics and classification techniques that will aid the practitioner in determining which tool is best for their given use case in Chapter 3. I elaborate on the Fidelity classification and classify each of the surveyed tools. I then introduce key

questions and challenges in the field of emulation and re-hosting, breaking down challenges into Pre-Emulation, Emulation - Setup and Execution, and Post Emulation in Chapter 4.

1.4 HQ-Tracer

When re-hosting a firmware using QEMU [2] or HALucinator [1], you will often get to a point where you are unsure of what is happening or why the system is crashing. Using a debugger, such as GDB, can be time consuming and, as the case for the base emulator QEMU, these debugging tools may not have reverse debugging as a general feature (at least not without manually creating checkpoints and such). PANDA [25] and other record and replay tools can be helpful, but all of this is done in the terminal which makes it difficult to visualize what code is actually being performed. Chapter 5 introduces HQ-Tracer that uses HALucinator/QEMU traces to visualize and see state inside of Ghidra. This allows the user to determine what is actually happening with a highlight view of execution and register state while seeing the disassembled and decompiled code in Ghidra.

1.5 PMatch

Chapter 6 introduces PMatch, a Ghidra plugin that performs *library matching*. This library matching is done at the Ghidra Pcode level, leveraging the intermediate representation to disambiguate binary matching at the strictly byte level (Pcode is Ghidra's register transfer language that is general in a way that it can model the behavior of essentially all processors). It still uses hashing for the main matching technique allowing for fast execution, but gives more flexibility than plain binary diffing tools, as it allows for partial matching as well as the ability to change the matching algorithm. On the other hand it is not necessarily as comprehensive as the control flow or data flow graph isomorphism approach, but it strikes a balance between the two approaches, balancing speed and flexibility. I introduce the motivation and show examples of what the RE process may look like when the practitioner has performed function matching versus not performed function matching. I dive into the implementation and discuss the limitations and benefits of using such an approach for function matching and show an example of how to use the plugin.

1.6 GHALdra

Chapter 7 introduces Ghidra HALucinator Emulator or GHALdra, which integrates High Level Emulation and different Re-Hosting Support Layers into Ghidra. As mentioned in the thesis statement, this tool enables emulation with the least amount of effort for the greatest number of ISAs when compared to current tools and approaches. GHALdra is implemented as a plugin with an Emulator Helper, an Intercept and Callback Master, a Peripheral Server and models for peripherals and breakpoint handlers. I describe the design and implementation of GHALdra and show an example of how it can be used and provide real output.

1.7 Additional Solutions

To aid in overcoming emulation challenges that are not strictly overcome by using GHALdra, I implement and provide tools to integrate with Ghidra and GHALdra that are discussed in Chapter 8. To aid in RE, I provide scripts to do graph creation and visualization, including dataflow and control flow, binary searching for context switching, entry-points, and functions of interest, string analysis and renaming in the Ghidra window, and loop analysis that can be helpful when looking for hardware polling loops.

1.8 VxWorks Re-Hosting Support Layer

In the VxWorks chapter, Chapter 9, I describe how adding a Re-hosting Support layer can enable emulation for Special Purpose embedded systems, particularly for VxWorks, which accounts for the largest number of non-linux embedded device operating systems today. I go over the background of why to emulate this type of system, and introduce the VxWorks system model, emulation utility scale and the actual re-hosting support layer. This work was enabled by work done on HALucinator in conjunction with Sandia National Laboratories [26], which I then tweaked to work with GHALdra.

2. BACKGROUND

The goal of this chapter is to introduce relevant techniques and tools that are popular in the emulation, firmware re-hosting, and binary analysis areas. This is not to mean that the tools mentioned or surveyed are all encompassing, rather they are closely related to the research I have performed and are the most relevant in my opinion. Because one of the main reasons/benefits of firmware re-hosting and system emulation is vulnerability discovery and security analysis, many of the tools focus on vulnerability discovery. Given this, I mention a few related vulnerability techniques that are integrated into some of the tools.

2.1 Analysis and Reverse Engineering

In the binary vulnerability and analysis field, there are two general techniques for any work: static and dynamic analysis. Static analysis is done without executing any lines of code, and is complete (if we say there are no vulnerabilities, then there are none at run-time) for the algorithms that it uses. If we didn't have the aliasing problem there would probably never be a need for dynamic analysis, but alas we are not so lucky. Because of aliasing and the limits of static analysis, dynamic analysis is growing increasingly popular. This is partially because of the ever increasing compute power and speed of emulation tools. Dynamic analysis is when we do an analysis with, or on, executed instructions. This analysis is usually not as "complete", but is often more useful when trying to understand what is actually happening in a binary. I will argue that dynamic analysis is not very useful without some form of static analysis before-hand, as understanding dynamic traces in firmware requires some static knowledge of a binary.

This leads to the general field of reverse engineering (RE) binaries, which is typical when researching malware, vulnerabilities, and cyber security in general. While these terms are ubiquitous for all types of hardware/software systems, I will focus on the assumption we are looking at embedded systems.

Staples of binary static analysis include function start search and identification, data identification, references and cross references, subroutine references, stack and argument analysis, string analysis, encryption analysis, base address analysis, external entry analysis,

disassembly and decompiler analysis, and address table analysis among a plethora of other analysis. Even with all of these analysis that are very useful, many times we have code pointers and aliasing problems, meaning that static analysis can only get so far. By using just a little bit of dynamic analysis we can further our RE efforts and continue iterating with static and dynamic analysis. Static analysis is essential for emulation to be able to get an accurate enough understanding of functionality for anything in emulation to be useful.

2.2 Emulation

Emulation of embedded systems is an emerging technique to accelerate discovery and mitigation of vulnerabilities in embedded system firmware. Embedded system emulation has traditionally been used during development to allow embedded software to be written and tested without the need for hardware. In cases where hardware is concurrently being developed, is costly to have in quantity, or is susceptible to damage, emulation is an appealing option. Just as emulation can be used to verify system behavior during development, it can be used for vulnerability research and analysis. Emulation provides the ability to deeply observe and instrument firmware in ways that are not possible on physical hardware. It can help analyze what operations are actually being performed at a lower level than static analysis of either the high level source code or even the binary level, and is a useful tool that is the basis for a host of vulnerability discovering techniques [1], [27]–[34] among other uses outlined in Chapter 2.

For a practitioner to use emulation or re-host[1] a firmware, there is a learning curve and a plethora of tools available. My primary purpose is to provide an end-to-end guide to the practitioner for firmware re-hosting along with tools that will work with a wide variety of Instruction Set Architectures (ISAs). I limit my focus on emulation and re-hosting to the embedded space, as virtually any system will have embedded devices, from wearable IoT devices to power plants, which if vulnerable will affect the entire system's security and functionality. I present an overview of current techniques/tools for the practitioner along with classification categories and techniques for evaluating which tool is best for the

[1]↑Re-hosting specifies that a binary that would run on a specific hardware is instead run on a host system using system emulation, and is therefore "re-hosted"

emulation task at hand. I provide tables for the practitioner to reference in subsequent challenge sections that specify whether a tool attempts to address the presented challenges, giving a starting point for the practitioner to evaluate the right tool to overcome challenges.

2.2.1 Evolution of Emulation

Since before the era of personal computers, emulation has been a technique used to broaden hardware use and increase simulation speed. For example, most printing software used to be designed for HP, so many non-HP printers would write an emulator to re-host the software designed for HP [35]. Re-using HP software allowed for faster time to market for new printers and reduced the development time and effort for creating new products.

Emulation theory was first developed in the early 1960s, with the 7070 Emulator for the IBM System/360 series of mainframe computers being the first implemented emulator [36]. This emulator allowed IBM's customers to continue running their existing applications after upgrading their hardware. As was the case for the 7070 emulator, early uses of emulation were to avoid obsolescence and increase hardware compatibility with the limited available software. Over time, manufacturers started creating hardware emulators to allow software development before the hardware production; decreasing product development time. Emulators are now becoming a popular tool for security analysis and logic debugging [29]–[31], [37].

Around the same time frame, simulation was also used, but it allowed for executing and expanding systems beyond what existed. Simulation is sometimes referred to in scientific modeling and investigating, but in this context, simulation is another technique to model the internals of a system. In the computer science context, simulation is modeling a system with implementation of the internals, whereas emulation is modeling by replacing some of the internals of the system. By replacing some of the internals, emulation can sometimes reduce complexity or increase firmware re-hosting speed. Emulation will often allow for original machine code to run directly on the emulator. Beyond the computer science context, simulation and emulation are sometimes used interchangeably, with simulation sometimes refering to a system being replaced by software. For our case, the distinction is not important, but the tools I survey mostly refer to their techniques as emulation.

One of the early emulation successes was Bochs [38], which was released in the early 90's. It emulated the underlying hardware needed for PC operating systems development, which enabled completely isolating the OS from the hardware. This isolation enabled restarting the emulator instead of reconfiguring hardware during OS development. Bochs was originally commercial licensed but was open sourced in 2000. In addition to Bochs, many other emulators were created including DOSBox [39], FX!32[40], and PCem[41]. These solutions were mainly geared for x86 or PC emulation.

As multiple emulators emerged, the *execution and memory fidelity* (*i.e.,* how closely the emulated system matches the real system, sometimes referred to as accuracy) varied from high (cycle and register accurate) to low (module and black box accurate). I discuss fidelity and give more classification points in Section 3.3, but fidelity and emulator speed are perhaps the greatest distinguishing factors between different emulators.

2.2.2 Emulation Bases

In the early 2000's, **Simics** [42] was created and evolved to emulate multiple architectures including Alpha, x86-64, IA-64, ARM, MIPS (32- and 64-bit), MSP430, PowerPC (32- and 64-bit), SPARC-V8 and V9, and x86 ISAs. It was originally developed by the Swedish Institute of Computer Science (SICS) before moving to Virtutech and eventually working its way to Wind River Systems, who currently sells it. Simics is designed to have fidelity at the instruction level, allowing for interrupt testing to occur between any pair of instructions. It also provides configuration management, scripting, automated debugging (forward and reverse), and other built in static and dynamic analysis tools to help with constructing an emulated system. One popular use for Simics was during the DARPA Cyber Grand Challenge to automatically vet submissions to check whether the submitted binaries adhered to the competition infrastructure [43].

In contrast to Simics and emulators that work at the instruction level fidelity, **QEMU** [2] gives up some accuracy to improve emulation speed. Instead of working as a sub-instruction simulator (performing multiple actions per instruction), QEMU execution and emulation occurs at the basic block level (sequential, non-control flow instructions), by translating

entire blocks of instructions to the host system's instruction set and executing the translated instructions. This allows QEMU to work much faster, as it does not have to check for interrupts at each instruction, and caching of blocks greatly reduces translation overhead. Because of its open-source license and community, QEMU has become one of the staples in academia and industry professionals. It emulates the IA-32, x86, MIPS, SPARC, ARM, SH4, PowerPC, ETRAX CRIS, MicroBlaze, and RISC-V architectures, and provides peripherals for many systems, making it and Simics two of the most widely used emulators.

One of the newest emulators available is **Ghidra Emulator**. Ghidra [44] is an open source software reverse engineering tool developed by the National Security Agency (NSA). The initial release in March 2019 contains emulation tools that allow for traditional software reverse engineering and emulation to be combined into the same environment. Because of the richness of features in these tools, there has been a large user base since its release. Ghidra uses their own processor modeling language called Sleigh and an intermediate register transfer language called P-code, with each machine instruction translating to up to 30 P-code instructions. This implies that the Ghidra Emulator works at the sub-instruction level (multiple emulator instructions performed per machine instruction), giving a relatively high execution fidelity as a base. Ghidra currently supports various existing architectures including X86 16/32/64, ARM/AARCH64, PowerPC 32/64, VLE, MIPS, MSP430, Z80, AVR, etc. To add a new architecture is simple in their framework, with the user only specifying how the new architectures instructions are dissassembled into the intermediate P-code language. Ghidra includes loaders, disassemblers, decompiler and analysis tools with the base of supplied analyses written in Java. Beyond the built-in emulator, there have been Ghidra P-code emulators emerging that allow for partial re-hosting of firmwares [45].

2.3 Related Vulnerability Discovery Techniques

As emulation has been used more frequently for vulnerability discovery, it is worth mentioning some closely related vulnerability discovery techniques that often leverage emulation. The techniques introduced are integrated into some frameworks that work with the base emulators, requiring at least some familiarity during discussion. I do not go into extensive depth

in these areas, but rather overview the techniques briefly, and recommend further reading for an in depth review. I mention some of the tools that are more popular, but it is not necessary to understand their differences or techniques in the scope of this review, rather the techniques are mentioned briefly in subsequent sections when integrated into a tool.

Symbolic Execution [46] is where symbols representing arbitrary values are supplied as inputs to a program (similar to letters in algebra representing numbers). The goal of symbolic execution is to analyze a program to determine what inputs can cause different parts of a program to execute. Rather than analyze and follow a single path with concrete values, a symbolic execution engine will use symbols to describe all program execution paths that can execute by using constraints on symbols.

Concolic Execution is when the tool will switch between using symbolic symbols and concrete symbols (like an algebra symbol having a set value, eg x=5) during emulation or execution. When reverse engineering firmware, an analyst will occasionally want to determine under what conditions a program will execute a certain portion of code. Symbolic and Concolic execution are tools that help solve this problem among other vulnerability discovery uses. Symbolic execution is now a common software testing practice, even though it was introduced in the '70s [47]. Commonly used symbolic and concolic execution tools include [27], [48]–[64].

Fuzzing is an automated vulnerability/bug discovery technique where random inputs are provided, and the system observed for undesired behavior (*e.g.,* crashes). There are many challenges to fuzzing and various tools that try to address these challenges. In the context of emulation, fuzzers often leverage the visibility into, and control over, a binary's execution to optimize their random inputs to improve their exploration of the binary. Some popular fuzzing tools that can be integrated with emulation are [65]–[89].

2.4 Surveyed Works

Prior to introducing emulation comparison axes, I introduce surveyed works that enable emulation and firmware re-hosting. I do not discuss or reintroduce Simics, QEMU, or Ghidra Emulator, as they are described in Section 2.2.2, but they are the base emulators used in the

tools introduced in this section. Base emulators can either run in user mode emulation (*i.e.*, running only user level applications) or full system emulation; with full-system emulation currently being the primary mode used for firmware re-hosting. Full-system emulation will emulate the processor as well as hardware peripherals; however, the set of emulated peripherals available in the base emulators are small compared to the diversity of hardware found in embedded systems. Much of the work in firmware emulation aims at solving this lack of emulated peripherals.

Avatar^2 [90] is a *dynamic multi-target orchestration and instrumentation framework* that focuses on firmware analysis. This tool was created by the same group as Avatar, but is a multi-target orchestration tool that has completely been re-designed from the original Avatar implementation [91].

The main contribution of this tool is that it allows various other tools (angr, QEMU, GDB) to interact and transfer data. Using these tools it can enable hardware in the loop emulation, where portions of execution or memory accesses are carried out by physical hardware.

angr [27] is a symbolic execution engine, that has been integrated with Avatar^2 to enable combining symbolic execution and other analysis into the concrete base QEMU emulator. angr is a binary analysis framework that provides building blocks for many analyses, both static and dynamic. angr provides an architecture library, an executable and library loader, a symbolic execution engine, built in analyses, and a python wrapper around a binary code lifter. It is actively developed and used in various other academic works including [65], [92]–[99]. I include this as one of the surveyed works because of its ability to use base emulators while providing new approaches and solutions to overcome emulation challenges from beginning to end.

HALucinator [1] addresses the challenge of providing peripherals not implemented in the base emulator by observing that the interactions with peripherals are often performed by a Hardware Abstraction Layer (HAL). It uses Avatar^2 and QEMU as bases to intercept HAL calls and replace them. They do this by manually providing replacements for HAL functions that execute during re-hosting. It uses a library matching tool to identify the HAL functions in the firmware. The tool relies on the assumption that the majority of

firmware programmers use Hardware Abstraction Libraries when writing firmware, which in our experience is a relatively safe assumption.

PANDA [25] is an open-source platform that builds on top of the QEMU whole system emulator that is used for architecture-neutral *dynamic analysis*. The main advantage of PANDA is that it adds the ability to record and replay executions, allowing for deep, whole-system analyses. System record has partially been addressed in a hardware-software co-design approach [100] as well as under more restrictive assumptions, using a purely software approach [101], [102]. Such whole system record and replay is challenging especially considering the timing requirements. PANDA, using the QEMU base and abstracting some of the analyses, allows for using a single analysis implementation across multiple computer architectures while maintaining the speed of QEMU, allowing some timing challenges to be addressed.

Muench2018, titled "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices" [103] demonstrate that tools used for desktop vulnerability discovery and testing do not necessarily transfer to the embedded space. In their paper they present different techniques used for vulnerability assessment implemented by instrumenting an the emulator used for re-hosting the firmware. They implement segment tracking, format specifier tracking, heap object tracking, call stack tracking, call frame tracking, and stack object tracking by combining PANDA and Avatar^2.

Using QEMU as a base, both **Firmadyne** [104] and Costin Firmware Analysis (referred herafter as **CostinFA** [21], [28] "A Large-Scale Analysis of the Security of Embedded Firmwares" and "Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces"), will extract the filesystem from a given firmware and re-host the filesystem on their own kernel. They perform system emulation using only software, not involving any physical embedded devices. Their tools work only for software images that can natively use chroot with QEMU. They then perform static and dynamic analysis on their re-hosted firmware to report vulnerabilities. Recent efforts that use the same type of approach as Costin and Firmadyne is ARMX [33]. ARMX requires more user interaction and configuration and only works for ARM architecture devices while requiring the rootfs from

the firmware and the extracted NVRAM from the firmware. Given these extra requirements, the results of the emulated devices using their technique is usually of high quality.

PROSPECT [105] and **SURROGATES** [106] enable emulation by forwarding hardware and peripheral accesses to the actual device; a technique known as hardware in the loop. PROSPECT forwards peripheral and hardware interactions through a normal bus connection to the device, but allows for analysis and implementation without needing to know the details about the peripherals and external hardware connected to the system. SURROGATES, in contrast, uses a custom low-latency FPGA bridge between the host PCI Express bus and the system under test, allowing forwarding and state transfer to and from the system's peripherals with transfers much faster than the original Avatar [91] system.

P^{2}IM [107] uses a drop-in fuzzer (AFL[66]) to provide inputs to their base QEMU emulator. They abstract peripheral and hardware IO and then use the fuzzer for providing the feedback to the base emulator. Their approach is different from existing emulation approaches as it does not use hardware or detailed knowledge of the peripherals, as the fuzzer provides interactions. The fuzzer enables executing firmware with simple peripherals to be emulated, but its ability to enable emulation of firmware processing data from complex and stateful hardware is unknown.

In contrast to using a random fuzzer, **Pretender** [108] attempts to re-host firmware by using machine learning to provide models of hardware interactions. Their system will record hardware interactions and all accesses to memory mapped input and output (interacting with peripherals is done through a specified memory-mapped address - MMIO) regions along with any interrupt that occurs during execution before performing a peripheral clustering and dividing the recordings into sub-recordings for each peripheral. They then train a memory model, trying to select and train on known models for each peripheral. The analyst then decides how to introduce inputs into the system.

In summary of the tools that are surveyed (highlighted in bold throughout Chapter 2 and Section 2.4) the general purpose emulators surveyed that have a wide variety of specialties and uses include Avatar^2, CostinFA, Firmadyne, Ghidra, HALucinator, Muench2018, PANDA, QEMU, and Simics. Emulators that use hardware in the loop include Avatar^2, SURROGATES, and PROSPECT. Emulators that can use symbolic execution and/or fuzzing

include angr, Ghidra, HALucinator, and P^{2}IM. The only emulator that uses machine learning for models is Pretender, which at the moment to be successful requires fairly simple firmware. If the reader is anxious for a flow-chart of which tool to use, refer to Figure 4.3 that is explained in Section 4.7. While this is not an exhaustive list of tools or methods, as there are many more available including [16], [89], [109]–[128] among others, I believe the scope of tools and their applicability is encompassed in the tools mentioned above.

3. COMPARISON TECHNIQUES

Now that I have introduced the works that have been surveyed, this chapter focuses on different comparison metrics and classifications. I first introduce axes over which the comparison of different emulators and tools can be compared. Various emulators and emulation techniques are used for different purposes, and thus make different design decisions. Establishing common axes to evaluate emulators is necessary to enable the practitioner to do a useful comparison before choosing the ideal emulation tool for their use.

3.1 Emulation Techniques

A significant challenge to re-hosting firmware is the large expanse of hardware peripherals that need to be emulated; thus, one evaluation axis is the fundamental technique the emulator uses to provide these peripherals. The technique employed will directly relate to another axis of comparison – the complexity of the hardware that is feasible to emulate. Some systems are simple with no peripherals, whereas others may be connected to Remote Terminal Units (RTU), Programmable Logic Controllers (PLC), Field-Programable Gate Array (FPGAs), multiple sensors, databases, Human Machine Interface controllers (HMI), etc. The amount of hardware the practitioner wants to emulate will range from a single chip or sensor, all the way up to the entire large scale system. The amount of hardware and complexity of the hardware emulation is largely limited by the peripheral emulation technique used.

The main peripheral interaction techniques used include *hardware in the loop* (HITL), *learning*, *fuzzing*, and *abstraction replacement*. *HITL* will use the emulator to perform instruction execution, but if accesses are made to hardware peripherals they are forwarded to the actual hardware. *Learning* refers to using machine learning to provide hardware interactions, whereas *fuzzing* will use random generation to provide simulated hardware interactions. *Abstraction replacement* provides peripheral hardware functionality by identifying software abstractions within the firmware and replaces execution of these abstractions with its own implementations. Examples of surveyed works that allow HITL are SURROGATES[106] and Avatar^2 [90]. P^{2}IM[107] uses fuzzing, and Pretender[108] uses learning, whereas Firmadyne [104], CostinFA[21], [28], and HALucinator[1] use abstraction replacement.

3.2 Types of Systems

In addition to how hardware peripherals are provided, it is important to consider the type of system the emulator is designed to support. The range and capabilities of embedded systems ranges from large multi-processor systems running customized versions of desktop OSes (e.g., Linux) to low cost, low power micro-controllers running a few KB of code without an OS. The challenges and techniques in emulating these systems vary and may or may not translate from system to system. I reuse the classification presented by Muench et al. [103], though use our own names instead of numbers for the types of systems, splitting the embedded system types into three different classes, based on the type of firmware they execute.

General Purpose Embedded Systems (GPES): also known as Type 1 embedded system, use a general purpose operating system that is primarily used on servers and desktop systems. Examples include real-time Linux, embedded Windows, and Raspberry Pi. The operating systems are retrofitted for the embedded space, but retain many desktop level features, but with stripped down components, and are coupled with lightweight user space environments such as busybox or uClibc. Tools such as Firmadyne and CostinFA require the embedded system they work on to be Linux based systems and will only work on this type of system. Emulating these types of systems greatly benefits from the work done to enable emulation of desktop software and operating systems (*e.g.,* QEMU directly supports emulating the Linux Kernel).

Special Purpose Embedded System (SPES): (Type 2 devices from [103]) use operating systems specifically developed for embedded systems. They are often commercial products and closed source. Examples include, μClinux, ZephyrOS, and VxWorks. These systems are usually single-purpose electronics or control systems. Some of the features that distinguish these systems are that the OS and applications may be compiled separately and the system is not derived from a desktop operating system. Thus, many emulation techniques from the desktop space do not work, and emulation must start from scratch. Re-hosting these systems requires re-hosting both the kernel and user space. Also adding to

the challenge of emulating these systems is the fact that the separation between the Kernel and user space is often blurred.

Bare-metal Embedded Systems (BMES): Type 3 devices are embedded systems without a true OS abstraction that I refer to as bare-metal embedded systems (BMES). They often do not have an OS, or may include a light-weight OS-Library. An example is an Arduino system. In both cases, the application will directly access hardware and the OS (if present) and applications are statically linked into a single binary. Recent work [1], [107], [108] focuses on re-hosting these systems.

I find this axis of comparison useful as it helps to determine what emulation techniques an analyst should consider for a given firmware. However, in practice, classifying a system is not necessarily cut and dry. Rather the classification is a continuum on the embedded space. For example, an embedded system that started out using UNIX OS 30 years ago may have originally been classified as a GPES, but as the system morphed over time, the same system currently may now be better classified as a SPES.

3.3 Fidelity

Introduced in Chapter 2, *fidelity* is perhaps the most important comparison axis, but also the most difficult to quantify. The difficulty comes from limited ability to inspect the internal state of hardware, and is further complicated by the ability to compare states. In an effort to enable better understanding of fidelity, I classify fidelity along the conceptual axis of *execution fidelity* and data or *memory fidelity*. This enables comparison of the conceptual limits of fidelity on a 2D plane. Work by Costin [28] has a general classification of emulators that have kernels and applications. The classification I present here is more general, applies to both re-hosting and emulators, has more classification points, and is applicable to all types of systems – GPES, SPES and BMES. The fidelity classifications are from the perspective of the firmware (software), and whether the emulation "looks" and "acts" like real hardware. This implies that I do not need to differentiate between memory (whether DRAM, SRAM, Flash, etc.) that looks the same to the firmware, I just refer to this as internal memory. If

there is another driver required to use specific memory (such as an SD card), then I consider this external memory.

Execution fidelity describes how closely execution in the emulator can match that of the physical system. I bin techniques into the categories *BlackBox, Module, Function, Basic Block, Instruction, Cycle, and Perfect*, with execution fidelity increasing from BlackBox to Perfect. A system emulated with *BlackBox* fidelity exhibits the same (or sufficiently similar) external behavior as the real system, but internally may or may not execute any of the same instructions a real system would execute. Module fidelity provides fidelity at the module level. For example, Firmadyne replaces the original firmware's kernel with its own to enable re-hosting the original firmware. Thus some modules of the firmware are executed unmodified, and others completely replaced. Function level fidelity accurately models the system at the function level (*e.g.,* HALucinator replaces entire functions to enable emulation). Similarly, basic block and instruction level fidelity accurately emulate at the basic block and instruction level layers. Beyond instruction level is cycle level which faithfully emulates to cpu instruction cycle (*e.g.,* the gem5 [129] simulator). Perfect emulation means that emulation is exactly the same as it would be on the actual hardware, which to our knowledge no current emulator achieves.

I categorize data/memory fidelity increasing from the coarsest granularity to finest granularity as BlackBox, Internal Memory, Register, and Perfect. BlackBox fidelity means that the data externally visible to the system or external memory (*e.g.,* HDD or SDD) is the same (*e.g.,* for a given input I get the same output). Internal Memory means that the internal memory (*e.g.,* RAM) is consistent with hardware for a given point in execution. Register level fidelity means that both internal memory and registers are correct at the given execution fidelity and perfect means that all memory components work the exact same as the given system at the level of execution fidelity needed. In Blackbox, Internal Memory, and Register Memory levels, these classifications are usually for specific areas of interest in the firmware. This means that there is a blur between the classification points, as some sections of the firmware may be at the Blackbox level where the user does not care much about the internals, but in a few sections of high interest the firmware emulation may be at the Internal or Register Level fidelity. The Perfect Emulation level is on the scale but currently

is virtually unobtainable. Depending on the practitioner, Perfect Emulation level can mean Register level throughout the entire firmware, or it could mean everything is exact, down to the cache for every execution cycle. In Figure 3.1 I show how the most prevalent firmware emulation techniques fit into this classification framework.

3.4 Purpose of emulator

Of the surveyed research works in firmware re-hosting, the main focus points have been *Creating Emulators, Dynamic Analysis, Static Analysis, and Fuzzing.* Each of these focus points enables the emulator to answer specific questions. The purpose of emulators include vulnerability detection, enabling running legacy code, hardware replacement, development assistance, and system behavioral analysis. The purpose of the emulator directly influences the techniques employed, types of systems the emulator will work on and for, and determine the fidelity of the system.

3.5 Level of Control

Another axis for comparison related to the purpose of the emulator, is the ability to control the exploration in firmware and what the tool can/should be used for. Control may perhaps be thought of as another axis in fidelity, quantifying whether you can control emulation and what is actually executed during emulation. It also refers to the level of interaction available to the practitioner. For example, HALucinator enables interactive emulation making it suitable for building virtual testbeds, whereas P^2IM enables fuzzing and would not be a feasible tool for testbeds.

For visualization ease I do not combine fidelity and exploration, but it is important to note that some tools and emulators do not allow for controlled exploration. For example, if your emulator is just fuzzing everything from memory to inputs, it may have very high execution fidelity, and low memory fidelity, but it has almost no exploration customization to specify what to execute, it solely relies on a random generator. Randomization and fuzzing enable high coverage fuzzing and vulnerability discovery, but it does not give a clear picture of actual real possibilities of execution.

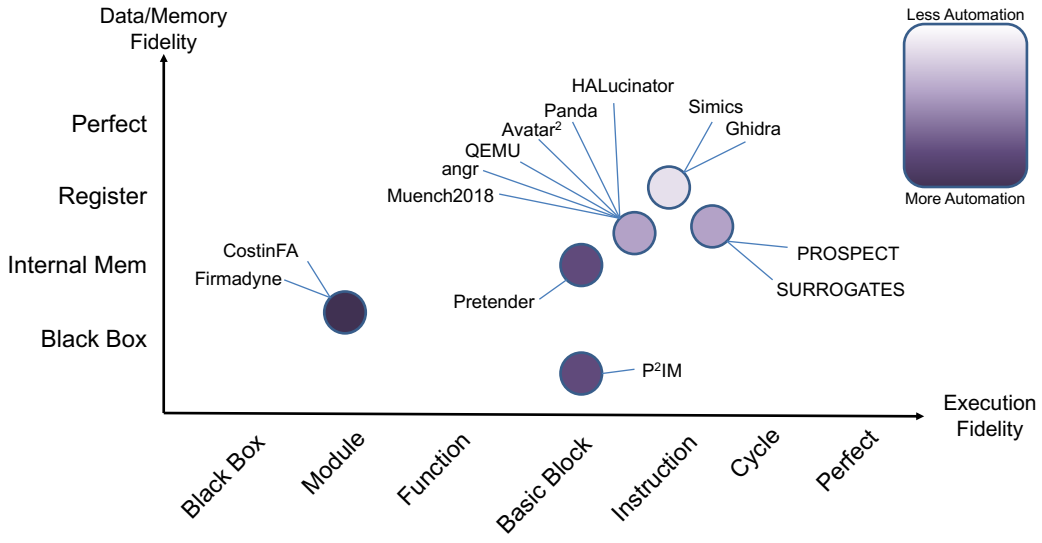


Figure 3.1. Categories of Fidelity

3.6 Classification of Surveyed Works

Now that I have discussed different axes of fidelity, I further categorize and compare surveyed works in firmware re-hosting and emulation. Figure 3.1 shows a 2D space of fidelity categorization. On the horizontal axis is execution fidelity and on the vertical axis is data fidelity. As can be seen in the figure, there is not a single point for each category, rather a blurry bubble, emphasizing that fidelity categorizations are conceptual and a continuum from black box to perfect fidelity. As can be seen in Figure 3.1, the points on the plot range from dark to light shading. The darker the circle indicates that the tools is more automated, requiring less user interaction and requirements for setting up the tool. The lighter the circle in the plot indicates that the default tool requires more user interaction to get the tool setup. I purposefully do not give concrete numbers to the automation or the fidelity levels, rather I visually categorize automation and place fidelity in regions to re-emphasize that these categorizations are not concrete and may slightly move/change depending on tool use and the practitioner's opinions.

3.6.1 Hardware In The Loop

As various works modify a core emulator, the fidelity can be improved or reduced. When the fidelity is improved there is often a trade off made in performance or complexity. SURROGATES uses QEMU as a base emulator, but adds specialized hardware to enable faster communication with real hardware. Specialized hardware allows for higher execution fidelity as the peripheral access is then perfect. HITL also increases the data fidelity, as there is no generalization for the peripheral model as is the case for PROSPECT. However, specialized hardware and HITL in general increases the complexity and cost of performing emulation while reducing scalability, as dedicated hardware is needed for each emulated system.

3.6.2 Instruction Level Execution Fidelity

I categorize Simics and Ghidra Emulator as instruction level execution fidelity while also having register level data fidelity because of their sub-instruction execution. While the coloring of the automation is the lightest of the works surveyed, there are ready made implementations of both of these emulators that can be copied or used out of the box to make the automation much closer to no interaction required. Yet, I color the automation at this level because of the tools defaults and to show the open ended-ness of using one of these emulators.

3.6.3 Basic Block Level Execution Fidelity

As mentioned previously, QEMU has basic block execution fidelity, and RAM fidelity as a default on the data fidelity axis. These are the default categorizations, and they can be affected by other tools. Muench2018, angr, PANDA, and HALucinator use QEMU as a base emulator without HITL, so in general their fidelity will be less than or equal to that of QEMU.

For any memory interaction, P^2IM will do random fuzzing, meaning there is not even blackbox fidelity for the data axis. Pretender uses machine learning to attempt to provide peripheral software replacement modules, reducing the data fidelity but still has some internal

memory correct and at least blackbox fidelity on the data axis. As these works reduce fidelity (on either axis) they also decrease the amount of effort needed for their emulation. Muench2018 and PANDA perform tracking/recording as mentioned in Section 2.4. While replaying and replication is enabled, the fidelity is still maxed out at the Basic Block and Internal Memory/Register fidelity because of the QEMU base emulator fidelity.

angr also uses QEMU for the emulation part of the tool, while also using symbolic execution. Symbolic execution is difficult to put on the emulation fidelity grid because it has multiple states as it executes, but by using QEMU for concolic and concrete execution, angr receives the same fidelity as the QEMU base. In contrast, Avatar^2 enables the interaction between virtually any of the tools, allowing the fidelity to be at any of the tools fidelity categorization depending on what is programmed by the analyst, but I show its fidelity point defaulting around QEMU, angr, PANDA, and HALucinator because they are default tools that are easy to get running in the Avatar^2 framework.

3.6.4 Module Level Execution Fidelity

Firmadyne and CostinFA lose both execution and data fidelity from their base QEMU emulator as they only extract the filesystem and run the code through their own kernel. However, by doing this they drastically increase the automation of their tools, making mass analysis of firmware more scalable.

4. QUESTIONS AND CHALLENGES

This chapter focuses on you, the analyst wanting to re-host a firmware, and discusses the challenges that are encountered, along with techniques and tools that can be used to address them. The challenges and tools surveyed provides a reference for the average practitioner or a starting point for new researchers in the area of firmware re-hosting and system emulation.

4.1 Questions of Purpose and Value

Each emulation tool has requirements that must be met before being able to re-host firmware. Some tools will bypass common emulation challenges with the technique they use. Ideally the tool will overcome challenges automatically, otherwise the practitioner must do so manually. Common preconditions are discussed in Section 4.4, however this discussion covers the challenges faced and not necessarily why they are encountered.

Before deciding to emulate a system or re-host a firmware, an analyst has a question they want answered. The idea of why to emulate a system is key to building emulators and is often not emphasized in emulation papers. When an analyst wants to find vulnerabilities, there are vulnerabilities that may be discovered at each fidelity level. It is therefore important to know what types of vulnerabilities you are looking for, e.g. is it a firmware logic error that can be detected by just correctly emulating the memory, or is it an instruction or hardware bug that needs higher execution fidelity?

Before emulating a system it is also important to do an analysis on the estimated cost and value-for-money evaluation of getting the emulation to work correctly and the tools to be used to perform emulation. To emulate some systems it may take a small team of engineers 6 months to a year to build and test for the desired system. This may be a bargain or may be too expensive. Analyzing the tools available and how to use them can speed up or even change the traditional approach of building an emulator (manually reverse-engineering the hardware, re-hosting firmware until failure and incrementally adding functionality).

4.2 Key Research Questions

I wish I could say that you should research one specific area, but for every solution/tool that currently exists, there are shortcomings. To understand what is *key* to your research, it is necessary to understand what you would like to solve with emulation. The key area I see that needs addressed is speeding up the time to emulate a system – this includes tools from all the subsequent sections to overcome the challenges. A script written for one tool may work to address a specific challenge, but that script needs to be ported if using another tool, *e.g.,* a script for Ghidra to help find the entry point will not necessarily work with angr or Simics.

This *key* research area of speeding up emulation really encompasses research across all types of embedded systems (GPES, SPES and BMES) and includes new ways of overcoming problems presented in Section 4.4, Section 4.5, and Section 4.6. Any tool or technique that more efficiently addresses the challenges is a useful area of research.

4.3 Challenges

Now that I have introduced surveyed works and classification criteria, I present some of the core challenges faced during firmware re-hosting and system emulation. During the emulation pipeline, various challenges are encountered and I split these challenges into:

- Pre-Emulation
- Emulation
- Post Emulation

Pre-Emulation are challenges that are pre-requisites to emulation execution and overcoming these challenges enables executing the first instruction in the emulator. These challenges include obtaining firmware, unpacking the firmware and gaining and understanding how to configure the emulator to re-host the firmware. Once the first instruction is executed in the emulator, I consider the Emulation stage to have been started, however, as execution progresses, greater understanding of the firmware is obtained, which leads to refinement of the

emulator configuration and implementation. Thus, I break Emulation into Emulator Setup — challenges that enable further refinement of the emulator and more complete execution — and Emulation Execution — challenges fundamental to emulation itself. Finally after emulation there is the Post Emulation stage where the execution is analyzed and validated. The challenges presented here have been mentioned in part and used as the foundation for multiple industrial and academic works, but I present them in entirety here for completeness.

In the tables throughout the rest of the paper, I survey different techniques that attempt to overcome the various challenges faced during emulation and binary analysis. If there is a check-mark, then the tool attempts to address the issue; but, it does not necessarily solve the problem and some tools work better than others, though I do not specify my opinions on the matter. If there is a dash, that means the tool bypasses the challenge by the technique they employ. If there is no mark or dash, then the tool does not address the challenge. In some cases the tools have prerequisites to work correctly. The emulator may require the analyst/user to figure out some of the challenges manually and pass the solution to the emulator to get past the challenge.

I provide some of the flow and common challenges faced during emulation in Figure 4.1, and Figure 4.2 though the figures do not encompass all the challenges that may be faced, with some challenges left off the figures all together for brevity. These figures are general flows and each bubble represents one or more challenges encountered through the process. The different challenges are further addressed in detail in Section 4.4, Section 4.5, and Section 4.6.

4.4 Pre-Emulation

In Figure 4.1 I show the flow that a practitioner will usually explore during the Pre-Emulation stage of re-hosting a firmware. With each step indicating challenges that need addressed before continuing to address the next. The challenges presented here are similar to those presented by Costin [21]. Here I focus on the challenges of emulating and analyzing a single firmware, whereas Costin focuses on challenges for acquiring and analyzing thousands of firmwares available to download from the Internet automatically.

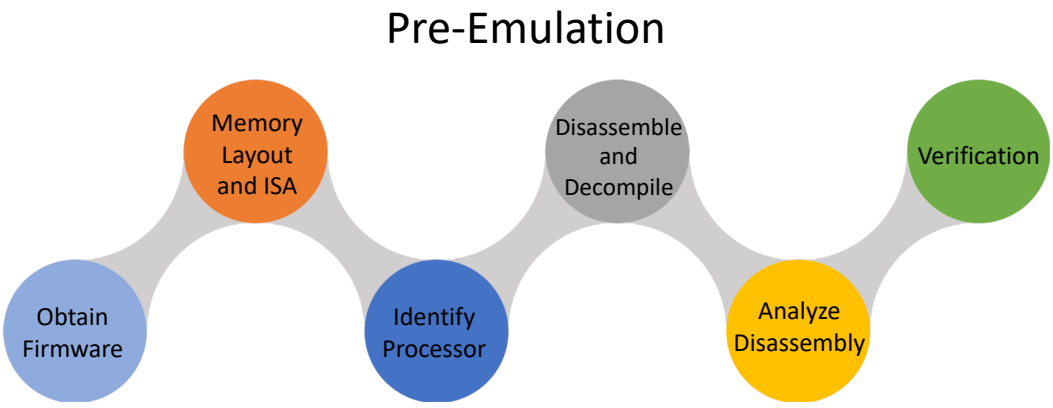


Figure 4.1. Categorization and flow of some of the steps required for system emulation during Pre-Emulation.

Before re-hosting a firmware, an analyst or practitioner will usually have a system they want to emulate or a firmware they want re-hosted. In some cases the system architecture is unknown or the analyst may not even have the firmware (*e.g.,* when working on a bug bounty or proprietary hardware). Even after obtaining the system or firmware in mind, key information must be identified prior to re-hosting a firmware. Steps and challenges prior to setting up the emulator is what I refer to as the Pre-Emulation stage. This stage may also include verification of information and understanding gained prior to actual emulation (*i.e.,* Disassembly, Initial Analysis, and CFG Recovery), though verification is not strictly necessary. I note that verification of correctness (which can include formal verification and/or behavioral verification) is useful when challenges in subsequent phases are encountered, as it narrows down where certain problems stem from.

The information required to begin emulation varies by emulation technique but includes obtaining firmware, determining memory layout, figuring out the instruction set architecture (ISA), identifying the processor, analyzing the binary, lifting/disassembly of firmware, and an initial firmware analysis. As a quick reference to the practitioner, different techniques that attempt to address the challenges present in Pre-Emulation are summarized in Table 4.1.

One of the main tools used to address these challenges is binwalk [117]. Binwalk is mostly used for extraction of the content from firmware images, but has other features that are useful. It can try to determine ISA (not necessarily the processor), extract files from a blob, do a string search, find signatures such as common encryption formats, disassemble using capstone [130], calculate entropy, and perform binary diffing. Some of the other tools are mentioned in the subsequent subsections. A simple example of reverse engineering from a blob that mentions some of the challenges below was presented at BlackHat 2013 [131]. They present an end-to-end unpacking of firmware with some of their formats that may be helpful to reference for a newcomer in the field. Another helpful reference for IoT firmware based on OpenWrt is maintained by OWASP. They provide a platform to educate software developers and security professionals on vulnerabilities in IoT devices [132].

4.4.1 Obtaining Firmware

The first challenge to re-hosting firmware is obtaining the firmware. In the simplest case it can be downloaded from the vendor's website. If not directly available from the vendor, 3rd party sites, such as Github, have firmware available that has been used for academic research. Other ways to get firmwares include obtaining example firmwares from development boards that can be compiled with various operating systems and toolchains.

Downloading the firmware is not always possible, and even when possible, the firmware can have embedded proprietary file formats that are not easily extracted. For example, files can be compressed or contain firmware for multiple architecture files with final compilation of the firmware being done on the hardware during boot-loading. In some cases, firmware is combined with the operating system, such as the case of BMES or SPES types of systems, whereas for others downloaded firmware is only the user level application and the operating system kernel also must be obtained separately to perform full-system emulation as is sometimes the case for GPES. To overcome embedded unpacking issues, sometimes a network capture tool (*e.g.,* [133]–[144]) connected to the actual hardware may be used to capture network traffic during a firmware update. The firmware is then extracted from the captured packet payloads.

In addition to downloading, firmware can sometimes be extracted from hardware. Vasile et al. [145] in their survey of hardware-based firmware extraction techniques showed a high percentage of systems expose UART interfaces that are sufficient to obtain firmware dumps. In addition to UART ports, debug ports (*e.g.,* JTAG), and USB ports can be used to dump firmware [146], [147]. On some devices, a physical acquisition my be achieved by using the flash memory read command after reverse engineering the firmware update protocol in the bootloader [148]. If these ports are locked and secured, another option is to remove the memory from the circuit board and connect to another system to dump the firmware. Removing memories from the printed circuit boards comes at considerable risk of damaging or destroying the hardware. Using debug ports requires the debug port be present, and unlocked, both of which are increasingly considered poor security design.

Of the tools I surveyed, Firmadyne and CostinFA obtained firmware from vendor websites, whereas P^{2}IM, angr, and Pretender used firmware available from a 3rd party vendor (Github). HALucinator and Pretender also used development examples from real embedded boards in their evaluation. For the case of Simics, QEMU, and Ghidra they are base emulator tools and not academic papers, hence they do not specify how to obtain firmware, expecting the user to have a firmware before using the tool. The other surveyed works did not specify how their firmware samples were obtained, just what the firmware was, or what the system was that they did emulation for.

The authors of Firmadyne developed and released a scraper that crawls embedded system vendors websites and downloads any firmware that they can identify. They then unpacked the firmwares in a generic way and if the firmware unpacked correctly, Firmadyne would continue with emulation, otherwise it would crash. Of the 23,035 firmwares Firmadyne scraped, they extracted 9,486 of them. Of those extracted, 1,971 were successful in naive emulation. The large percentage of failure (90%) shows obtaining and extracting firmware is a real challenge and is difficult to overcome in many circumstances. CostinFA posted URL publicly that would try to unpack and analyze a firmware [149]. CostinFA collected an initial set of 759,273 files scraped from publicly accessible firmware update sites, and filtered that down to 172,751 potential firmware images. A sampled set of 32,356 images were then analyzed and 38 vulnerabilities were discovered [21], though it is unclear the level of emulation or re-hosting involved. Other resources that may be helpful in obtaining firmware include Python scraper tools and other open source repositories [150]–[153]

4.4.2 Instruction Set Architecture

After obtaining the firmware, it is necessary to determine what the instruction set architecture (ISA) the firmware uses so that the emulator can disassemble the firmware into the correct machine instructions, endianness (*e.g.,* little- or big-endian), and word-size. In addition to determining the ISA family (e.g ARM, PowerPC, X86, MIPS, ARM64, AVR, etc.), sometimes the ISA version is needed to correctly disassemble instructions *e.g.,* is it ARM with Thumb support and floating point instructions or not?

The ISA can most commonly be determined from a datasheet of the processor being emulated. If the hardware is unknown and a datasheet cannot be identified, static analysis techniques can be used. These techniques first try to determine if the file format is of a known file format (*e.g.,* ELF, PE2, Mach-O) using the file utility, then looking at other signatures in the firmware (*e.g.,*, encryption, compression, etc.), or analyzing strings in the binary to guess the ISA. One well known tool that may help determine the ISA is binwalk [117]. Binwalk will use the capstone disassembler and try disassembling the binary for various types of ISA. If there are more than a specified number of instructions in a row (default 500) of a given architecture, then that is a strong candidate for the ISA. Of the tools surveyed, Firmadyne and CostinFA use binwalk and existing extraction tools to determine the ISA and then extract the filesystem, or extract the filesystem first and then determine the ISA – with the order of these two steps being highly specific to the firmware and analysis process at hand. Ghidra has headless scripts that can be run to try and determine the ISA (including one using binwalk), whereas angr provides the *boyscout* [154] tool that will try to determine the architecture and the endianess of the firmware. The other surveyed tools expect the ISA to be given to work correctly.

There is also recent research on using machine learning to classify ISA and endianness. These techniques usually rely on doing a binary similarity detection. They will use known architectures for training binaries, decomposing the binaries into smaller comparable fragments and transforming them to vectors to work with machine learning and stochastic gradient descent optimization methods. One such work by Clemens et al. [155] experiments on 16,785 different firmwares from 20 different architectures, and was accurate in classifying 17 of the architectures over 99% of the time, and the remaining 3 architectures (mips, mipsel, cuda) 50% of the time. De Nicolao et al. [156] leverages supervised sequential learning techniques to locate code section boundaries of binary files to help ease difficulty in analysis using a similar sized database for learning. Kairajärvi et al. [157] uses the ideas from both Clemens [155] and Nicolao [156] with improvements while using and publicly releasing a much larger and more balanced database for learning and testing. Other machine learning techniques that could be modified to determine the ISA are mentioned in[158] along with other machine

learning techniques for binary analysis. These techniques require large datasets for training to be accurate and will only work as well as their training data.

If the above automated methods fail, the practitioner can attempt to manually identify the ISA by looking at signatures and strings in the firmware (certain compressions, signatures, copyrights, etc.) and/or brute force decompilation and performing an "eye" test on what looks promising.

4.4.3 Determine Base Address

In order for the firmware to execute in an emulator, it must be loaded at the correct address(es). Determining the base address is difficult if the firmware being re-hosted is a binary-blob (e.g. just a binary with no symbols or metadata). The base address where the firmware should be loaded can sometimes be found by hardware datasheets (for firmware executing from internal memory). If source and compilation tools are available, base address information can be found in linker scripts.

Finding the base address is fundamental to many binary analysis techniques, and thus techniques to (try to) automatically determine it have been researched. Zhu et al. use strings and LDR instructions, comparing them and matching the offsets to determine the image base of the firmware [159], [160]. This is similar to the technique of listing the dwords occurring in a file with the list of strings in a file and lining up the distances between occurrences if possible. If there is a match, then subtracting the offsets will give you the image base address.

Firmalice [96] leverages jump tables in the binary by analyzing the jump table positions and the memory access pattern of indirect jump instructions. In a jump table there are a set of absolute code addresses which can give a better idea of where the firmware needs to be located at for the absolute addresses to work correctly. To find the jump tables, they scan through the binary for consecutive values that differ in their least significant two bytes. Finding the jump table is successful in many cases as jump tables are typically stored consecutively in memory. After finding the jump table, they then analyze all the indirect jumps found in the disassembly phase and the memory locations that they read their jump

targets from. The binary is then relocated so that the maximum number of these accesses are associated with a jump table.

angr also attempts to determine the base address with their analysis script called *girlscout* [161]. This script will try to decide the base address by looking at functions and doing a statistical analysis to vote on the most likely base address. If automatic techniques do not work, the base address may be discovered by brute force guessing and checking, as most base addresses are multiples of powers of two.

In the case of Firmadyne and and CostinFA, the filesystem from the firmware is extracted after determining the ISA, and then the filesystem is used in a custom kernel given to QEMU. By using their own kernel, these tools bypass the need to determine the base address, but it also reduces the execution fidelity, as it does not execute the original kernel. For the other surveyed tools it is assumed the base address is provided by the practitioner.

4.4.4 Finding Entry Point

After determining the base address of the firmware, the practitioner needs to determine the entry point (*i.e.,* address from where to start execution). Entry point information can be encoded in the binary (*e.g.,* Executable Linker Format, ELF defines the entry point in metadata), or different analysis can be run to help give the practitioner entry point options.

For binary-blob firmware, angr, Ghidra, and IDA[124] have scripts that scan through the binary attempting to find function prologue instructions and function returns. From function information a directed function call graph can be generated and analyzed. Any root node of a weakly-connected component in the call graph can be treated as a potential entry point. Function call graph creation and analysis requires that you already know the ISA (so the function prologue, function epilogue, and call instructions can be analyzed). The call graph technique often returns multiple entry points, from which the correct entry point for emulation must be identified. Additionally, firmware will often have multiple valid entry points (*e.g.,* bootloader and interrupt service routines).

Instead of looking at the firmware to identify the entry point, some techniques rely on knowledge of the hardware they support to determine the entry point. For example,

HALucinator targets ARM Cortex-M devices. The Cortex-M architecture defines that the initial program counter's value be stored at address 0x4 on reset. Thus, HALucinator finds the entry point by looking at address 0x4. If the hardware is known, the datasheet will likely provide information about how the system begins execution.

For techniques that use a replacement kernel, as is the case for CostinFA and Firmadyne, finding the entry point challenge is essentially bypassed, as the entry point of the kernel is known because they built the kernel themselves. If the practitioner is dealing with a known operating system or compiler toolchain, the entry point function name may be specified (such as _start or _init). There are then techniques that can do function matching giving you the actual entry point. One such example is VxHunter[162] that will work for many firmwares with the VxWorks OS and compilation toolchain.

4.4.5 Determine Memory Layout

Determining the memory layout enables configuring where in the processors layout different types of memory are located. It sets the address for RAM, Flash, and MMIO. If the system is available with specifications, usually the memory layout can be determined from datasheets. For ARM systems, there may be a CMSIS-SVD file that also defines the memory layout. These files are in a specified format that can be loaded into a reverse engineering (RE) tool such as Ghidra or IDA and an automatic analysis can be run to update the memory layout [163]. Other types of files that sometimes specify memory layout include EDC files and hwconf files.

If no documentation exists and RE tools fail, physical examination of the components on the device that is being emulated may help to determine memory layout. Often markings and manufacture printing on the parts can be used to identify datasheets for the parts. If the memory layout is still unknown, the emulator can be over provisioned with memory (e.g., giving significantly more memory than the physical system) and trial and error must be used to determine where code is located and where external peripherals are, then these interactions are usually mapped into memory mapped IO. Of the surveyed tools, memory

layout must be specified by the practitioner, though in some cases tools make it easier *e.g.*, Ghidra when CMSIS-SVD file is present.

4.4.6 Identify Processor and/or Board Support Package (BSP)

Depending on the fidelity the practitioner is targeting, often times identifying the exact processor is not necessary. As the case is with QEMU, the emulator only works off the ISA level features. For fidelity at the cycle level of emulation, tools such as gem5 [129] require knowing the exact processor as the same ISA instruction can be implemented differently at the cycle level for various processors. In QEMU, if a versatile machine is not used, there can also be errors, even if the ISA specified is correct.

In the case that the practitioner needs to solve the challenge of identifying the processor, solving memory layout may help. However, if the processor is known, often the documentation for the processor will specify the memory layout. If you have access to the hardware, most likely the processor will be labeled, solving the challenge.

If you do not have access to the hardware, the practitioner could do an aggressive instruction finding, followed by an analysis on how the instructions interact with memory. This analysis can then be compared against analysis on known processors to narrow down the processor candidates. This is still a manual process for the practitioner to then select and test different processors. If the vendor is known, projects that leverage known information from other reversed firmware may be used [164]. If the previous analysis are unsuccessful, the practitioner could run a brute force script to test all available processors in the base emulator. If still not successful, access to labeled hardware is required. For surveyed works based off QEMU, identifying the processor is not strictly necessary in all cases, though if the QEMU target is not versatile it may be required.

4.4.7 Disassembly, Initial Analysis, and CFG Recovery

As mentioned at the beginning of Pre-Emulation, disassembly, initial analysis, and recovering the Control Flow Graph (CFG) are not strictly Pre-Emulation challenges. However, it is important to verify that the work the practitioner has performed and challenges addressed

to this point in time are correct. Verification of previous work is perhaps skipped in some pipelines, but I have found that verification before continuing on to the next stages will save time and reduce headaches in the Emulation phase, *e.g.,* when multiple possible entry points (which may occur when there is a bootloader in addition to the main firmware under analysis), doing a validation and verification that you have the right entry point is worth the extra time to save the practitioner from wasting time re-hosting the bootloader that may not answer the emulation question at hand.

In the case that the practitioner has an idea of what the firmware is doing, such as is usually the case when the hardware is known, analyzing the CFG can help determine if the previous challenges such as base address, ISA, entry point are correct. When the CFG is recovered, it also conveys how well the disassembler and decompiler performed. This will give an idea of how successful emulation execution will be and/or the fidelity of emulation moving forward. Of the works surveyed, there are relatively few core disassemblers. angr uses capstone for parts of its disassembler, whereas QEMU and Ghidra use C/C++ to implement their disassembly. The other surveyed tools use a base emulator that will reuse one of these disassembly implementations. If the disassembler works perfectly, then the instructions have been disassembled correctly and the CFG can be recovered trivially if the control flow does not leave the main processor.

If the control flow leaves the main processor and goes to a co-processor (*e.g.,* GPU or DSP) chip for initialization, it is near impossible to recover the CFG for what occurs in the co-processor. The practitioner at this point can try to determine what state and memory changes occur by comparing the state of memory before and after control is passed to the co-processor. If control flow leaves the emulated processor, fidelity of emulation is limited for that portion of the re-hosting, but in most cases is not a limiting factor in emulation.

If the hardware is not known and the practitioner does not have any idea of the control logic the firmware is trying to perform, disassembly and analysis can still be of use for validation. Disassembly will ensure the correct ISA is known and possibly help validate the processor. A control flow graph can still be created and analyzed to determine if there are valid flows throughout the graph, giving at least a weak reassurance that the base address and entry point are correct.

Disassembly and analysis are iterative, meaning the practitioner will perform the disassembly, then analyze the results in iterations. Between iterations, they will then modify the inputs and parameters to the disassembly slightly depending on the analysis results. This process will continue until the analysis results match what is expected from the practitioner. During this iteration process, there will be tweaks to the ISA, processor, base address, entry point, memory layout, etc. To solidify the ISA, processor and memory layout, the practitioner may analyze multiple firmwares for the same system concurrently, aggregating the analysis results to solidify results.

4.5 Emulation

In Figure 4.2 I show the flow that a practitioner will usually explore during Emulation setup and execution stages of re-hosting a firmware. The stages will usually be interspersed during iterations of emulator development. These challenges generally are not linear, but rather occur in different orders depending on the firmware being re-hosted.

After the practitioner has determined the processor, memory layout, entry point and base address of the firmware, verifying support in the base emulator is performed. Ideally the base emulator has support for the processor, if not, the practitioner will have to create a new specification for the base emulator. QEMU and Ghidra have instructions on how to add support for a new processor [169], [170]. After a base emulator is available, the next challenges can be addressed.

Overcoming the challenges in pre-emulation enables loading the firmware into the emulator and beginning execution. For the execution to be faithful to the real system, additional challenges must be overcome. I break these challenges into sub-categories– setup and execution. Setup challenges are usually done statically when the emulator is either paused or stopped whereas execution challenges are when the emulator is actually running. As a reference to the practitioner, I provide Table 4.2, but do not provide a reference table for the execution, as all the surveyed works attempt to overcome the given challenges.

Table 4.1. Pre-Emulation Challenges

Paper/Technique	Obtaining Firmware	Determine ISA	Finding Base Address	Finding Entry Point	Determine Memory Layout	Identify Processor	Disassembly/ Recover CFG
angr [27]		✓	✓	✓		✓	✓
Avatar² [90]							✓
CostinFA [21], [28]		✓	—	—	—	—	✓
Firmadyne [104]	✓	✓	—	—	—	—	—
Ghidra [44]		✓	✓	✓	✓	✓	✓
HALucinator [1]				—			✓
Muench2018 [103]							✓
P²IM [107]							✓
PANDA [25]							✓
Pretender [108]							✓
PROSPECT [105]							✓
QEMU [2]							✓
Simics [42]							✓
SURROGATES [106]							✓
ARMX [33]		—	—	—	—	—	—
BANG [116]		✓	✓	✓	✓	✓	✓
BAT [165]		✓	✓	✓	✓	✓	
Binwalk [117]		✓	✓	✓	✓	✓	
CLIK on PLCs [16]							✓
Datasheets		✓			✓	✓	
Dtaint [166]							✓
Dytan [118]							✓
FIE [119]						—	✓
Firmalice [96]		✓	✓	✓			✓
firminsight [150]	✓	✓			✓		
firmware-mod-kit [120]		✓	✓	✓	✓	✓	
firmwaredb [151]	✓						
FirmUSB [121]						—	✓
HumIDIFy [122]		✓	✓	✓	✓	✓	
ICSREF [123]		✓	✓	✓	✓	✓	✓
IDA-PRO [124]		✓	✓	✓	✓	✓	✓
Inception [125]							✓
KLEE [59]							✓
OFFDTAN [167]							✓
PIE: Parser [168]							✓
Radare2 [115]		✓	✓	✓	✓	✓	✓
Scraper (python)	✓						
subzero [164]	✓	✓					
Spedi [126]		✓	✓	✓	✓	✓	✓
strings (linux command)		✓				✓	

✓means the tool attempts to address the given challenge. '—' means the tool bypasses this challenge, usually by the emulation technique used. A blank means the tool does not address the challenge. The first 14 entries in the table are bolded and are the main tools profiled in detail throughout.

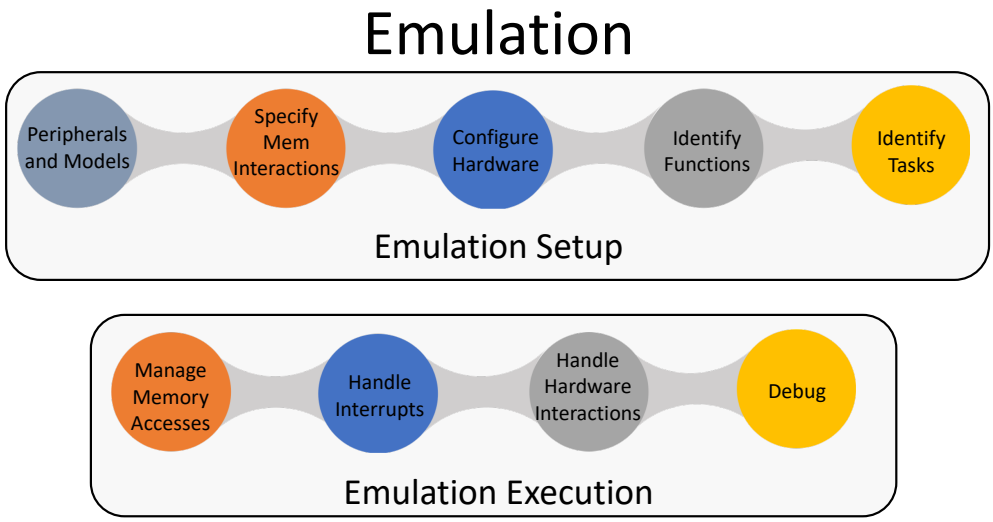


Figure 4.2. Categorization and flow of some of the steps required for system emulation during Emulation Setup and Execution

4.5.1 Emulation Setup

After you have overcome the challenges in Pre-Emulation, now you need to determine how to handle configuration, external interactions and memory. Emulation setup encompasses these problems and is closely tied with the actual emulation execution. Setup is often iterative between execution stages, and with each iteration more knowledge is gained about the firmware and its dependencies on the emulator and peripherals. The emulator is improved and execution of the firmware is also iteratively performed. This continues until the question the analyst was trying to address has been answered.

As mentioned in Section 3.1, the scope of emulation may vary from a single chip or micro-controller to a subsystem or a large distributed system. The scope of emulation that is targeted will affect the challenges faced during emulation. The problems I discuss here are not all encompassing, though I believe it is a sufficient basis to demonstrate major challenges currently faced during the Emulation Setup phase.

Peripherals, External Hardware, and Modeling

Handling peripheral accesses is where a large amount of research is currently focused. Because of the variety of peripherals and vendors, it is likely the peripheral that is being accessed by the firmware is not implemented in the base emulator. Handling External Hardware and Peripheral interactions encompasses how to handle or represent interactions between the emulator and the peripherals. As mentioned in Section 4.4.5, statically determining the memory map will specify where the peripherals are located, not if they are used or when they are used. Dynamic analysis and execution can provide some of this information, discovering what peripherals are used when valid inputs for the given subset of peripherals analyzed. Depending on the fidelity of the peripheral model, i.e. the extent to which the peripherals are modeled, will constrain the dynamic execution to specific paths. Providing more realistic peripheral models will increase the amount of code executed and result in valid accesses to more peripherals that may not otherwise be accessed. During dynamic execution, if the emulator has an exception or crashes due to trying to access peripherals that are not mapped, this usually occurs when there is an aliasing problem. Solving the alias problem

is sometimes feasible statically, but may have to be solved dynamically during execution by keeping track of various addresses and using this information to update peripheral models for the emulator.

Handling peripheral interactions can be done by emulating the external device, by using the actual hardware (HITL), or by patching the firmware to bypass the interaction. The firmware modification may ignore some of these accesses (such as setting CPU clock rates), always give an increasing value, or read from a file of expected inputs. The necessary operation will partially be determined by the firmware being re-hosted. Using HITL will give the highest fidelity for execution and memory, but will prohibit the parallelization of emulation as it is constrained to availability of the connected hardware. HITL emulation also has significant challenges in synchronizing states between the emulator and hardware. For example, a timer may generate an interrupt on the hardware, causing it to be stuck continuously processing the timer's ISR, while the emulator has no timer and thus is not processing any interrupts, or another example is when the watchdog timer interrupt kicks in while the analyst is slowly emulating or using a debugger (such as GDB).

Providing abstractions requires manual effort and a thorough understanding of what the peripheral is performing, but it does provide high fidelity. Tools that use this approach include QEMU, HALucinator, PANDA, and Muench2018. Using a fuzzer, on the other hand, does not require device specific knowledge but may not give sufficient fidelity for many questions you want answered and is usually only useful to help find bugs and vulnerabilities. P^{2}IM uses the fuzzer approach and is successful with finding some vulnerabilities. Using machine learning, as Pretender does, is appealing as fidelity is perhaps slightly affected but could still give valuable insights beyond simple software bugs. The results of current machine learning approaches only show proof of concept on simple peripherals (*e.g.,* serial/UART port being the most complicated) at this point, and it is still unclear if the approach will work for arbitrary hardware peripherals.

Of the other surveyed tools, Firmadyne and CostinFA assume the peripherals are part of their core kernel, otherwise the emulator will crash. They do not provide techniques around such challenges. SURROGATES and PROSPECT will forward peripheral accesses to the

actual hardware, bypassing the modeling, but it introduces the state synchronization and delay of emulation challenges.

Memory Interactions and Setup

Most emulators allow the practitioner to specify where data, code, and peripherals are located. Doing so allows the emulator to set restrictions on emulation, such as crashing or notifying the practitioner if data is trying to be executed. Emulation setup will usually allow specifying different memory regions and types of interactions such as where RAM, Flash and Memory Mapped peripherals are located. This emulation setup can also be described as configuring the emulator, which is necessary to start the emulator. Setup will include providing the information discovered during Pre-Emulation, including specifying the base address and entry point along with the memory layout and amount of memory available.

If a tool does not have built in support for memory, reusing the base emulator support or providing the interaction through software are feasible options for the practitioner. In specialized cases such as SURROGATES and PROSPECT, interactions can be forwarded to the actual hardware. For HALucinator, Ghidra, PANDA, PROSPECT and Muench2018 it is expected the practitioner will either use the built-in memory handlers or provide modules to allow memory interactions to perform correctly. For Firmadyne and CostinFA the memory and handling of memory is built directly into the kernel, so if there is an error with the memory or interactions that is not a firmware bug, their kernels will need to be modified.

Configuring Hardware

Setting up hardware is a challenge for emulation using HITL. This requires initializing it state such that its can be used, and then bringing it into the loop. To do this the practitioner will need to specify in the emulator how and when to forward interactions to the hardware. This may also require delays or using specialized hardware execute fast enough to be usable, as is the case for SURROGATES or Aveksha [100]. Aveksha is a system for nonintrusive tracing of execution at a high spatial and temporal granularity suitable for an embedded

wireless node. Avatar^2 is a powerful tool when configuring hardware, and the provided framework is closest thing to allowing plug and play integration for HITL.

Missing Code

In some cases, as you recover the CFG you may notice you have missing code. Missing code usually occurs when either the wrong entry point has been specified, there is patched out functionality that may write code (such as a bootloader), the firmware is not complete (*e.g.,* either a partial firmware-update or unpacking was only partially successful), or if there is code on ROM chips in the device. Missing code is more common if the firmware is ripped off the actual hardware device. In some cases it is possible to patch out the missing code and still obtain the level of emulation fidelity the practitioner cares about, otherwise, missing code may make emulation infeasible.

If the emulator tries to execute missing code, usually the system will throw an exception or crash altogether. The only technique I am aware of to overcome this challenge is replacing the code with models or skipping the code altogether. HALucinator replaces the code by using function intercept techniques that then allow for replacing such functionality with manually written models. For other surveyed tools, missing code will require some sort of manual intervention to overcome, perhaps manually providing the same functionality as HALucinator.

Function Identification and Labeling

Function Identification is necessary for some emulation techniques but not all. If emulation fidelity is at the module or function level, the practitioner may want to determine certain functions, such as Hardware Abstraction Library (HAL) calls, and provide abstractions for these functions. Function identification is not an easy problem and is the basis of a plethora of papers and is still a very active area of research [171]–[183].

Of the surveyed works, angr, Ghidra, and HALucinator have library matching built in to their frameworks. The other works either do not need to overcome such problems or they do not address function identification. angr, Ghidra and HALucinator will use existing

Table 4.2. Emulation Setup Challenges

Paper/Technique	External Hardware and Peripherals	Mem Interactions/ Setup	Configure Hardware	Missing Code	Function Identification and Labeling
angr [27]	✓	✓			✓
Avatar^2 [90]	✓	✓	✓	✓	✓
CostinFA [21], [28]	✓	✓	✓	✓	✓
Firmadyne [104]	—	✓	✓	—	✓
Ghidra [44]	✓	✓			✓
HALucinator [1]	✓	✓	✓	✓	✓
Muench2018 [103]	✓	✓	✓	✓	✓
P^{2}IM [107]	✓	✓	✓	✓	
PANDA [25]	✓	✓	✓	✓	
Pretender [108]	✓	✓	✓	✓	✓
PROSPECT [105]	✓	✓	✓	✓	
QEMU [2]	✓	✓	✓	✓	
Simics [42]	✓	✓	✓	✓	
SURROGATES [106]	✓	✓	✓	✓	
ARMX [33]	✓	✓	✓	✓	✓
Datasheets	✓	✓			
Firmalice [96]	✓	✓			✓
ICSREF [123]					✓
IDA-PRO [124]	✓	✓			✓
Inception [125]	✓	✓			✓
KLEE [59]	✓	✓			✓
OFFDTAN [167]		✓			
PIE: Parser [168]	✓	✓			✓
Radare2 [115]	✓	✓			✓
Spedi [126]	✓	✓			✓

✓means the tool attempts to address the given challenge. '—' means the tool bypasses this challenge, usually by the emulation technique used. A blank means the tool does not address the challenge. The first 14 entries in the table are bolded and are the main tools profiled in detail throughout.

techniques to try to identify some functions, such as using IDA FLIRT signatures, loading libraries and trying to match, and compiling existing HALs and comparing the re-hosted firmware to a database. Other techniques include identifying functions from their side effects such as is the case with Sibyl [184].

4.5.2 Emulation Execution

Execution deals with the classical problems of emulation as mentioned in [2], expanding to include problems that are arising with the increasing uses of emulation. The different fidelities of execution will also use different techniques for emulation. Tools that are cycle accurate, such as gem5 [129], will decode the instruction depending on the processor and use

the same depth and stages of the CPU pipeline as the original processor when emulating the instructions. Simics [42] has instruction level fidelity and will thus decode the instructions and update state after each instruction. QEMU uses basic block fidelity and does a translation from the basic block instructions of the target architecture to machine host instructions using QEMU's Tiny Code Generator(TCG). Ghidra Emulator will translate instructions to P-code, with each machine instruction translated generating up to 30 P-code instructions. Others will use various other intermediate representations including LLVM IR [185], VEX [186], REIL[187], BAP's BIL [112], Binary Ninja LLIL[109] and more. Of the surveyed tools, angr will translate to VEX. Each of these techniques have their own challenges and trade-offs, which is why there is still research in this area.

For each of the surveyed tools there is a base emulator. angr, Avatar^2, CostinFA, Firmadyne, HALucinator, P^{2}IM, PANDA, Pretender, PROSPECT, SURROGATES and Muench2018 all use QEMU as their base emulator. angr also uses CLE loader to allow Avatar^2 targets to run concretely in their framework through what they term angr_symbion that combines symbolic execution and concrete execution. Cross tool integration essentially opens the door for any emulator to work with any other analysis tool, symbolic engine, or fuzzer. For traditional execution challenges (the different subsections below), the base emulator will usually address and overcome the challenge. Because the base emulator solves the challenges throughout execution challenges, I do not specify what each tool does for overcoming the given challenge, rather I address more generally what QEMU, Simics and Ghidra Emulator do.

Register Allocation

Each architecture will have different registers and different conventions. For example, the program counter (PC) on different architectures will be different registers such as R15 for ARM, PC for x86/x86-64/PowerPC, R0 for TI-MSP430. Most emulators will map each register to a concrete fixed host memory address or register. Mapping the registers to fixed host memory is versatile and probably the most portable solution and is done for QEMU, Simics and Ghidra Emulator.

Direct Block Chaining

Block Chaining is directly related to QEMU, as Simics does a sub-instruction level emulation and execution, leaving the chaining to occur naturally with fall through if the instruction does not modify control flow. Ghidra will allow natural fall through, but allows for flow modification in their UI.

If the emulator does a translation, such as QEMU and the translation of entire basic blocks, then the emulator has a simulated program counter which is used to find the next blocks of code to be executed. These blocks are usually cached in memory to speed up execution, so there is a lookup in a hash table to retrieve the correct block. For some emulators, they will add instructions at the end of blocks to directly chain to the next block of instructions to execute.

Self-modifying code and translated code invalidation

On some CPUs self modifying code is not a problem, as there is a specific code cache invalidation instruction executed when code is modified. On other CPUs though, there may not be an invalidation instruction, so this becomes more difficult. In [2], they handle self modifying code by keeping track of translated code and the corresponding host page as read only. If a write is performed to the code, then they invalidate the translated code allowing for the code to be rewritten. They do some more clever things when using a software MMU, as they don't always have to invalidate the code when only data is changed. Ghidra Emulator has permissions for writing and code is read only. Therefore if code is written the emulator will throw an error and exit. It is not clear how or if Simics supports self-modifying code.

Non-Volatile Memory

Non-Volatile Memory (NVM) are becoming more popular because of the improved latency and power efficiency compared to flash and other hard drives. Traditionally NVM has not been a challenge faced as it is now becoming more prevalent. Because NVM is relatively new, on boards that have NVM, most current system emulators are probably lacking in

support. Some vendors give instructions on how to enable an emulated environment where you can build persistent memory (PMEM) applications without having the actual hardware. To overcome the NVM or PMEM challenges in system emulation, you need to determine how the memory interacts and provide a handler for it, much like QEMU does for normal memory. The instructions given by different PMEM hardware vendors on how to emulate their hardware will be crucial during implementation. QEMU, Simics and Ghidra Emulator handle NVM by allocating memory in the host system and having hardware interactions modeled by the analyst [188]. For QEMU, some tools provide some interactions such as NVRAM [189].

Direct Memory Accesses (DMA)

In some cases co-processors may use direct memory access (DMA) for initialization, or peripherals may write directly to memory. DMA can be addressed by emulating co-processors, using HITL emulation, or if the firmware uses known function calls for DMA, the functionality can be replaced by intercepting and replacing the functionality. QEMU has been expanded by Avatar^2 in software, and SURROGATES using hardware, to allow for forwarding peripheral acesses to the actual hardware. Simics is usually a more custom solution and also allows for HITL with given modifications. HALucinator will handle DMA by intercepting HAL calls that perform DMA and will implement the needed memory modifications with manually implemented functionality.

Handling Interrupts

There are various ways to handle interrupts. Handling interrupts is another active research area as there are multiple ways to accomplish the interrupt handling. The emulator may or may not check at each instruction or translated block for interrupts, with tradeoffs for different implementations. In some cases the emulator may require that the user trigger interrupts, call the interrupt handler, or patch an automatic call at certain parts in the firmware. There are pros an cons to each of these techniques. For the case of QEMU, they do not check at each translated block whether there is a hardware interrupt pending, rather

it relies on the user to asynchronously call a specific function and specify that an interrupt is pending. At that point the function resets the chaining of the current executing translated block, ensuring that execution goes to the main loop of the CPU emulation. In this main loop it looks to see if there is a hardware interrupt pending. By requiring user asynchronous interrupt calls and checking after basic blocks for interrupts, it allows the emulator to be much faster with less overhead while still supporting interrupts. Simics will allow for interrupts between each instruction by contrast, allowing for a higher execution fidelity at the cost of speed. Ghidra Emulator can also have interrupts at the sub-instruction level between different P-code operations.

In both cases, if you are testing something such as an interrupt storm (sending multiple interrupts with various orders and frequencies), you will most likely have to asynchronously specify interrupts to the base emulator. When those interrupts are handled will affect the fidelity of the system.

Multi-Threading

In some systems, multi-threading is enabled. Applications that use multiple threads may use locks/semaphores for inter-thread communication, but this requires that your system emulator allow multiple threads to "run" at the same time as well. If the system allows multiple applications to be run, they will employ a scheduler and use either pre-emptive or co-operative scheduling. With co-operative scheduling the emulator essentially just needs to allow the multiple threads to execute, and those threads will manage themselves during interaction. For pre-emptive, this will require trigging an interrupt to cause the scheduler to run and perform context switching between the threads. During emulation, usually the practitioner will start with single threaded applications, running one at a time and gradually increase the complexity and number of threads as the system gains more functionality and fidelity.

As mentioned throughout, with the ability to have fidelity at the instruction level, Simics is known for being more useful when debugging multi-threaded firmwares or systems. QEMU is able to emulate such systems, but in some cases it will give a false sense of correctness

because threads can only interact at the end of basic blocks for QEMU, whereas Simics and Ghidra Emulator allow thread interaction between any pair of instructions, even in the middle of a basic block. QEMU can attempt to overcome this limitation by setting breakpoints at every instruction, which essentially makes each instruction a basic block, but this dramatically slows down the system to the point that the emulation is not useful.

Debugging

During system emulation, undoubtedly you will run into some errors, whether they are errors in the actual firmware and/or system, or your emulator has errors and bugs. In either case, you need to be able to integrate some debugging. GNU GDB is a popular option to integrate into tools, having plugins for some of the major softwares including Avatar^2, angr, Ghidra, QEMU, etc. Another option I have found useful is first emulating a serial port or other form of printing messages, enabling "printf" style debugging. In addition to printing, directly inserting function calls into the firmware can be useful for debugging purposes.

QEMU works as well as the debuggers that connect to it, with logging and setting some break points which should be familiar for those who have used GDB. Ghidra Emulator also allows for a GDB bridge to be connected for debugging. Simics employs integrated debugger support with both forward and reverse direction debugging available.

Timing Constraints

If the purpose of your emulation is to answer questions regarding timing, you may be limited by the emulation approach and tools you use. In some cases you may be able to use a hardware integration solution such as SURROGATES, with specialized hardware to forward memory or peripheral accesses to an actual device. If you need to parallelize your emulation, HITL may not be feasible. In such circumstances, the practitioner may have to implement their own timing. Implemented timing can be cycles, instructions, basic blocks or something similar. Timing is in many cases closely related to enabling interrupts, as some interrupts are timing or watchdog interrupts and in those cases require some sort of triggering or timing implementation in your emulator.

If the base emulator is Simics, the emulator has a better idea of the number of cycles required for emulation, even if the emulator is slower to perform the operations. QEMU, in contrast, will be faster, but non-deterministic in execution time, which may be unsuitable if timing guarantees are needed.

4.6 Post Emulation

Post Emulation challenges are usually directly related to the purpose of emulation. Some of the challenges include determining/patching vulnerable code, comparing actual vs wanted behavior, finding vulnerabilities, visualizing emulation results, and automating the previous steps overcoming challenges.

4.6.1 Finding Vulnerabilities

Finding vulnerabilities is arguably the most popular reason for system emulation. There are many techniques to find vulnerabilities, with fuzzing being one of the most popular techniques.

There are many other techniques used to find vulnerabilities, including data flow analysis, taint analysis, control flow analysis, record and replay execution analysis, dynamic and symbolic execution. Some tools use supervised machine learning for vulnerability assessment, *e.g.,* Costin et al. [190] use ML to classify firmware to help both address firmware vulnerability discovery and vulnerable device discovery. With the vast amount of work in vulnerability discovery, in depth discussion and scope is not feasible in this paper, so I suffice to say these methods exists, and refer the readers to existing evaluation articles, including [191]–[193].

4.6.2 Verification

Once you have a system emulator, whether your first version emulator or a polished version, the problem of verifying that the emulator does what it is supposed to do arises. While verification of SoC chip functionality before taping and production of a whole circuit system is well studied[194], verification of emulated systems and re-hosted firmwares seems to be

lacking. None of the current tools verify whether they are correctly emulating the re-hosted firmware, beyond black box behavior comparisons. In addition, there are no real benchmarks or verification techniques that are standard to test emulators. With the expansion of firmware emulation as an area of research, verification of the emulators and of re-hosted firmware execution is a critical need.

4.6.3 Analysis

As with normal hardware/software systems, emulated systems also are tested and analyzed after "completion". The firmware used during emulation may be analyzed and tested for bugs. The emulator itself may be analyzed to help with hardening firmwares that run on actual hardware. Analysis and post emulation includes vast amounts of effort in binary hardening, vulnerability detection and other cyber security efforts. All of the surveyed tools do some sort of post emulation analysis.

4.7 Considerations

When determining what base emulator or the different techniques/tools to use, there are pros and cons to each solution. Some of the main considerations I contemplate before choosing specific tools include hardware support, fidelity, performance, debug support/availability, and usability/control. These considerations are subject to the practitioner's opinion, though I have tried to objectively classify the tools fidelity and automation as shown in Figure 3.1.

In Figure 4.3 I provide an overview for giving a decision flowchart on choosing a tool, though I leave out a definitive path when trying to narrow tool use based on debug support, usability and control, as I have found opinions vary greatly for the various tools. On the left of the figure in a big box are all the surveyed tools. Following the arrows from there gives the key difference between the emulators. If the analyst is interested in machine learning, Pretender will be a good place to start looking. Likewise, if interested in finding bugs/vulnerabilities using symbolic execution and fuzzing, then angr, Ghidra, HALucinator, and P^{2}IM are a promising starting points. If you are interested in using actual hardware along with emulation, Avatar^2, SURROGATES, and PROSPECT are appropriate. Otherwise, general

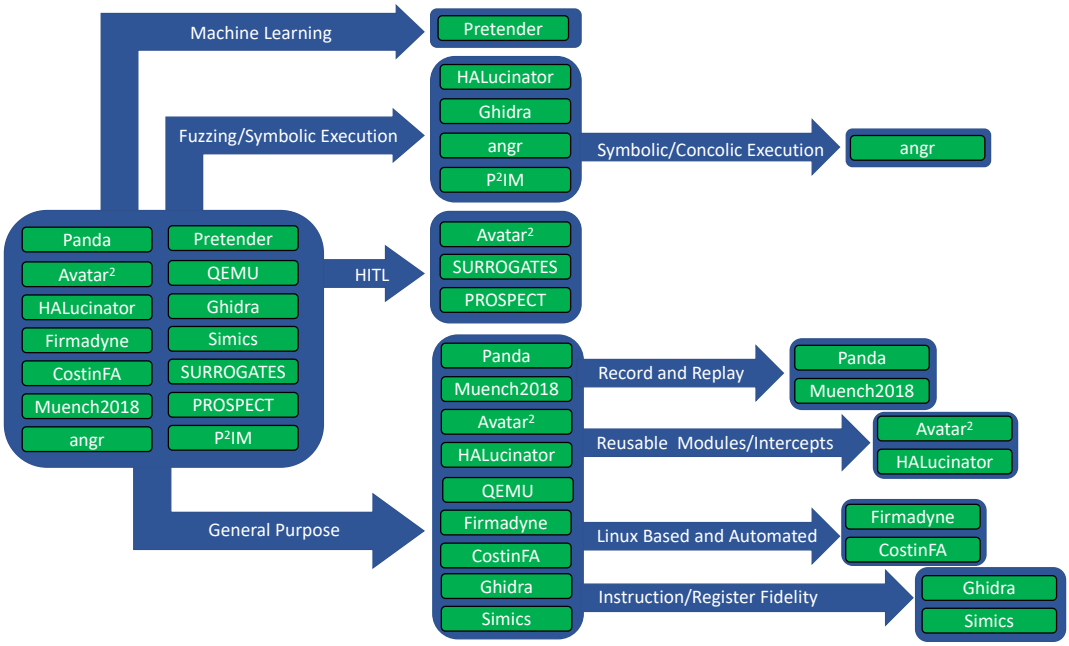


Figure 4.3. Flow Chart to Choose Emulator

purpose emulators that have different strengths and uses include PANDA, Muench2018, Avatar^2, HALucinator, QEMU, Firmadyne, CostinFA, Ghidra, and Simics. Once you have narrowed down the potential tools to a group of tools the following considerations should be evaluated.

Hardware support has perhaps been the most significant reason emulation has not been used in the past. The effort to emulate a system or re-host a firmware has traditionally been difficult to get working with systems (and some argue still is). This is slowly being remedied, with the core emulators having support for a wide variety of hardware. In addition, there is an increased effort to make adding new hardware and processors simpler and easier. One of the first considerations will still be if the tool you want to use has existing support for the hardware you want to emulate.

Emulator fidelity, as mentioned in Section 3.3, will narrow down which tools can be used to answer the question the practitioner is using emulation for. In some cases, symbolic execution is a feasible option to answer the emulation question at hand. In such cases, angr is the best option surveyed, being user friendly with a dedicated Slack[195] group with

help and development channels. If symbolic execution will not work, the practitioner can reference Figure 3.1 for help in narrowing down which surveyed tools can be used for the needed fidelity.

After considering fidelity, *performance* and *debugging support* also need considered. Depending on the emulation question, one of these attributes will be more important than the other. One of the main reasons emulation has not been used extensively in the past is because of the overhead and slow performance of emulators, but as software and hardware become more powerful, performance issues are being overcome.

When the performance of the emulator is a hindrance, partial emulation or specialized hardware are possible solutions. Ghidra Emulator allows for partial emulation, letting the practitioner start emulation at specific functions given they specify necessary inputs. SURROGATES provides specialized hardware to interact with memory faster. HALucinator sets breakpoints on user specified addresses provided in a YAML file. By analyzing only the necessary memory to answer a specific emulation question emulation performance can improved.

Debugging support is also a key attribute to consider. If the practitioner is re-hosting a firmware with limited knowledge of what the firmware is doing, more debugging support will be essential. As noted, one of the benefits of emulation is being able to examine memory at each point during the firmware re-hosting. Examining memory requires the use of debugging tools or logging. For base emulators, QEMU and Ghidra (through [196]) have GDB integration. Using GDB, break points can be used to step through emulation. Simics also has built-in debugging and allows for break points, along with stepping forward and backward through emulation. It is important to note that debugging support will affect performance, as the more break points and tracking needed will slow down the emulator. Both performance and debugging support are active areas in research, with base emulators trying to implement faster tools while monitoring and debugging.

Usability is a combination of exploration and control as referenced in Section 3.5 and tool automation. If a tool is automated, performs fast enough, and can answer the emulation question at hand, that tool is obviously the best option. However, at the moment, this is not a common scenario, and thus how much control the practitioner has with exploring and

executing re-hosted firmware is also an attribute that must be considered. Tools with GDB integration allow for expanded control. Tools that use fuzzing or randomization have less control and may or may not be useful for the practitioner in some cases. In essentially all the tools surveyed, debugging is built in to the base emulators and/or is available through plugins or default built in.

4.8 Summary

As a practitioner is contemplating the correct tool to use for firmware re-hosting and system emulation, I suggest using the provided emulation comparison techniques and considerations I have presented here. I have discussed comparing emulation tools by the techniques they employ, the types of systems they work for, the purpose of the tool, how much exploration the tool can achieve, and the most useful classification technique, fidelity.

Along with comparison techniques, I have shown classification of surveyed works including angr, Avatar^2, CostinFA, Firmadyne, Ghidra, HALucinator, P^{2}IM, PANDA, Pretender, PROSPECT, QEMU, Simics, SURROGATES, and Meunch2018. These techniques and their differences were discussed as I presented the core challenges faced during emulation and firmware re-hosting, giving an overview for the new practitioner or researcher looking on where to begin and further research in the firmware re-hosting and emulation field. Automation in current tools reduces the fidelity of the system dramatically, but in many cases is still sufficient for some emulation questions to be answered. The endeavor to automate current tools is difficult because of the wide coverage of embedded systems, but I note the valiant effort the surveyed tools have made and recognize the importance of making tools easier and automated for practitioners to use.

System emulation and firmware re-hosting has mutated and evolved dramatically since its inception. The challenges faced when trying to emulate a system/re-host a firmware have grown as emulation has improved. I have highlighted the need for further research and tools to address Pre-Emulation, Emulation, and Post Emulation challenges. As tools and solutions are invented and released, I have provided classifications and criteria on how to evaluate such tools.

5. HQ-TRACER

When trying to re-host a firmware, you will come to a point where you can boot the firmware in your emulated system, but you don't really know what is happening, let alone what is supposed to happen. Ideally, you want a tool that helps visualize what is happening, which leads naturally to a reverse engineering platform such as Ghidra or IDA. Because IDA requires a paid license and there is less open source contribution, I see Ghidra as the obvious choice for visualization.

The decompiler in Ghidra does well in most areas, and if it struggles on certain things, the user can tweak the decompilation rather easily. One of the most useful reverse engineering efforts in my opinion is function matching and decompilation. In Chapter 6 I go into more depth on function name matching, but even after you have matched and done some RE, static analysis only gets you so far. I know this might be blasphemy for some, but dynamic analysis combined with static analysis helps give a better understanding of what is actually happening in a firmware when compared to either static or dynamic analysis alone. To this end, I wanted to incorporate both static analysis performed into Ghidra, using the dynamic traces that occur in an emulator in conjunction with built-in Ghidra static analysis.

QEMU is probably the most widely used emulator base at the moment, so I looked to incorporate traces performed in QEMU into Ghidra. I see colossal benefit of using a technique such as HALucinator when trying to scale dynamic analysis. Because HALucinator also uses QEMU as a base emulator, I created a Ghidra plugin to visualize HALucinator/QEMU traces, hence the tool name HQ-Tracer.

The typical debugging that occurs with QEMU is done by using GDB to single-step execution. But QEMU does basic block translation and caching to speed up execution, which makes tracing what is happening difficult, especially if you think you are single-stepping when in reality you are actually executing multiple loops or blocks. This does not mean it is infeasible to single-step, rather it is not the default expected result for a more novice user.

Logging is another option for debugging with QEMU, with many options to log and print different things during QEMU execution. Using the output logs/traces from QEMU

Table 5.1. HALucinator Argument Passthrough Table

HALucinator option	Passthrough Argument
–singlestep	-singlestep
–log-blocks=exec	-d exec
–log-blocks=trace	-d in_asm,exec
–log-blocks=trace-nochain	-d in_asm,exec,nochain
–log-blocks=regs	-d in_asm,exec,cpu
–log-blocks=regs-nochain	-d in_asm,exec,cpu,nochain
–log-blocks=irq	-d in_asm,exec,int,cpu,guest_errors,avatar,trace:nvic*

Passing the command line argument to HALucinator in the left column corresponds to the Avatar/QEMU passthrough argument in the right column.

have a lot of information and are relatively straight-forward to integrate into Ghidra for visualization. If one would like to debug using the single-stepping using GDB, there are tools to integrate GDB with Ghidra, which highlight instructions currently being executed by GDB inside of a Ghidra window. This can be useful, but getting it setup with QEMU and GDB is difficult, and I, as a user, prefer the logging approach because of the ease and speed.

In HALucinator there are options that pass through logging options to Avatar[2] and eventually QEMU. In Table 5.1 I specify the arguments that can be passed to HALucinator in the left column and the corresponding translations to QEMU in the right column. I added the `regs-nochain` and `singlestep` options to HALucinator for more detailed logging, though in practice I usually use `--log-blocks=regs --singlestep`, and when more detailed information is required I will use `--log-blocks=regs-nochain --singlestep`. I will mention that the `-nochain` on any of the options makes the logs *much* larger and makes the logging and subsequent HQ-Tracer slower.

The `--singlestep` will break basic-blocks in QEMU into single instructions. This is especially useful when taking advantage of register tracking abilities in QEMU. Again this will make the logs larger, but it also enables the HQ-Tracer to have register state at each instruction in the trace.

5.1 HQ-Tracer Options

Above we see some of the options required to perform logging at the HALucinator/QEMU levels, but now I will introduce some of the options for the HQ-Tracer tool. I go into more detail on implementation in Section 5.2, but this section discusses what the options and commands are for the plugin.

HALucinator will keep statistics and information on all of the functions that it intercepts and put these statistics into a YAML formatted file. When running the plugin, the user is prompted whether the trace being loaded is a QEMU only trace, or if HALucinator is used as well. If HALucinator was used, the plugin will also highlight functions that were intercepted, even though not every instruction was executed. This is for visual effectiveness to show that there was execution occuring at that point in the program.

HQ-Tracer will load the logs into Ghidra and highlight all of the instructions in the trace in yellow. It then highlights the current instruction in green. This current instruction might be visualized as the execution point, even though emulation has already occurred and the execution trace is just being visualized. The HQ-Tracer commands that are available at this point are summarized in Table 5.2.

The normal forward 'execution' is available using the `n` command or the `s` command. The difference is that `n` will move forward by instructions in the trace, whereas the `s` will try to move forward in the trace to the next instruction in address order. This means that a branching or calling instruction will not be stepped 'into' when using `s`, rather it will try to go to the next instruction right after that branching. This makes it similar to GDB or other debuggers where the user can step into a function or step over a function call. Another forward command is the `o`, which attempts to continue stepping forward until the function changes. This can be at the end of a function or in the middle of a function if there is an embedded function call.

Backward 'execution' is also available, which is useful when you go to the end of a trace and want to track back to see how you got to a certain point. The `p` steps backward, doing a 'reverse' execution. Other options include searching for a specific address/instruction in the trace by a search next or search previous command.

Table 5.2. HQ-Tracer Command Options

Comand	Command Explanation
e	executes your command (useful for debugging this plugin) example: e print("Hello World")
h	help - print all the available commands
n	step over/next example: "n" will move forward 1 instruction in the trace. "n 10" will move forward 10 instructions in the trace.
o	performs the 'n' command until out of function. Perform the 'next' command until we leave this function Will stop at first function call or the return
p	step backward example: "p" will move backward 1 instruction in the trace. "p 10" will move backward 10 instructions in the trace.
q	quit - shutdown the plugin
s	Step Over - Get next instruction in address order, perform 'next' command until we reach this instruction If the next instruction is not in the trace it will fail.
t	Toggle Stack Tracing - enable/disable return address stack tracing
sn	search next (sn) - search for the specified address going forward. example: "sn 0x0" or "sn 0" will search for the address 0
sp	search previous (sp) - search for the specified address going backward. example: "sp 0x0" or "sp 0" will search for the address 0

Another useful tool, specifically for users wanting to extend or debug the tool, is the use of the `e` command. This command allows you to type any valid python/jython code in the command box and have it execute at the plugin/object level. There are experimental functions that can load multiple traces and highlight the intersection of the multiple traces, but this functionality is not a default capability. By using the `e` command the user could call these experimental capabilities, though it requires a more intimate knowledge of the plugin.

A heuristic return address stack tracer is also included, though it is not very accurate for multi-threaded traces. For traces that are single-threaded it performs relatively well, though our experiments are limited on the number of firmwares we tested on. Because of this limitation the functionality is turned off by default, though it can be enabled by using the `t` command.

5.2 HQ-Tracer Implementation

In this section I break down how this plugin was implemented, getting into more of the gritty details of things. It is important to note the default programming language of Ghidra is Java. They also provide a standalone Jython executable inside their deployment, which currently is version 2.7.2. This means that any Python code needs to work through the Jython interpreter. To do this there are usually a few options. It is possible to use the Jython version of 'pip' for installing many of the Python libraries, but you are usually restricted to packages that do not use any C compiling. This can be very limiting, and in such cases I have found it useful to look for Java libraries that attempt to provide the same functionality.

I implemented the plugin entirely in Python. I separated the functionality into 4 main files, namely the GUI, Plugin, Start, and Main. The GUI is straight-forward, using `javax.swing` and `java.awt` to import dimensions, constraints, buttons, labels, lists, panels, etc. In my opinion the GUI just needs to be intuitive enough to be useful and not too complicated to introduce bugs or inhibit use. To fulfill these very stringent requirements, I include a 'Start' button, a 'Quit' button, and a command text box that you can type commands into. There are labels that specify whether you have started the plugin, and it has a little prompt above the input text box that tells you to type 'h' if you need help with the commands.

When the trace has registers enabled, there is a box below the input command that will show the state of the registers at the end of the given basic block. This is why the `--singlestep` command is so important to enable on the logs, as it will give you the register state at each instruction when enabled.

The window for the plugin can be moved and docked anywhere in Ghidra, which is one of the benefits of using the GUI packages mentioned. In this GUI file, I have button handlers and a handler for the command line box that will pass the actions to the Plugin class. The Plugin class integrates the GUI and the brains of the plugin into Ghidra. It provides functions for getting the monitor, state, view, etc. from Ghidra. It also has commands to update the GUI view and sync the highlighting/coloring inside the Ghidra Listing and Decompile windows. The Plugin class will take the text from the command line GUI, initialize the plugin

if it hasn't previously been started, and call the command execution handlers inside the Main file. Lastly, it provides the functionality to remove the Plugin from Ghidra, releasing all state and returning resources.

The Main file requires YAML to work with HALucinator. In my release I include the PyYaml required files for `import yaml` to work, though the practitioner can use any PyYaml version >= 5.2 and it will work correctly. The PyYaml is used to read the HALucinator stats file. During initialization, the Tracer will register the current Ghidra state, making sure that there is a valid open file to be worked on. It will initialize a list of instructions with an empty register state and set the current execution point to be 0. It initializes all of the command handlers and will ask the user to input the name of the log file. I prefer to have an empty line that I can copy and paste the complete path of the file into, so I chose this option for the input. If a user prefers a prompt to pop up and let them navigate the filesystem to select the file it is as simple as changing one function call name in the Main from `askString` to `askFile`.

This is where the implementation details of QEMU and Ghidra collide and don't work as well. When printing the register state, QEMU will print the registers with different names and capitalization than Ghidra. For example, for an ARM v4-Thumb ISA, QEMU will print out 'R00', 'R01',...'R15'. In contrast, when getting the registers from Ghidra, it will return 'r0', 'r1', ...'r12', 'lr', 'sp', 'pc' along with other registers (such as 'r13_svc', 'r14_svc', 'spsr_svc', etc). For this reason, when matching registers from QEMU to Ghidra, I rely on user specified registers, as the logic required for the > 30 ISAs to parse the differences from QEMU to Ghidra is not realistic for this plugin.

Inside of the QEMU log parsing, I check to see if the line is an address, part of the register state, or something else. Because the stack tracking and register tracking logic is a little more complicated, I will first explain what happens just with the addresses, then branch out in my explanation. During parsing of the log, if the line in question is an address I will save the address into a Ghidra address set and a list for indexing and highlighting. When adding register tracking, usually the register state in QEMU is multiple lines in a log file. Because of this, there are flags to determine if the previous line we read was also part of the register state. We gather all of the register state for the given block in the trace together, and save

this state, linking it to the instruction in the indexing list (explained in the simple case). When adding the stack tracking logic, we check whether the given instruction is in the same function as the previous instruction, or if it is different, and then assign an `add`, `delete`, or `nothing` action. The action we determine is linked to the corresponding instruction in the indexing list. If the new instruction is in a different function than the previous instruction, we then inspect the current function stack. If the function that contains the new instruction is currently on the stack, we attach a `delete` action. If it is not on the stack, we attach the `add` action. If the new instruction is in the same function as the previous instruction, we add a `nothing` action. This is where we run into trouble with heuristics though, as some functions will share code, for example, a context switch and exit task. Depending on Ghidra and how the compiler chain implemented compilation, these may show as 2 different functions, or a single function. Another issue that may affect heuristics is a tail return, where some of the functions will stay on the call stack even when they should be removed at a return.

In the Main file is code to update the highlighting and execute each of the commands mentioned in Table 5.2. Because we did most of the algorithmic work in the loading and initialization, the commands that we have will essentially index into our list and display the stack and registers associated with the instruction while highlighting the current index in green.

The final file is a very short start script that can be run from Ghidra's script manager. To run the plugin, the user will download the source files; then inside your Ghidra environment enter the script manager (shortcut by using the green and white play button on the toolbar), and click on manage script directories button (this is 3 lines towards the right of the window). The user will then press the add button and navigate to where the source code is located and press OK. To start the plugin after it has been enabled, the user will run `hqtrace_start.py` from the script manager. This design is simpler than the default plugin and extension enabling required. This start script will pop-up the actual plugin window. The window is modular and like other Ghidra windows can be docked anywhere inside the environment or stay stand-alone. Visual examples are provided in Section 5.3 below.

5.3 HQ-Tracer Example

Because of the nature of binaries and intellectual properties, I will show examples on open source firmware that is released with permisive licensing. This means that the example shown is simple and can be redistributed in source and binary forms with or without modifications as long as I provide the license copyright notice with it. For this example I am using an STMicroelectronics STM32469I-EVAL firmware. The hardware has a SYSCLK that runs at 180 MHz for the STM32F4xx device. The UART peripheral in the hardware allows data transmission of a predefined data buffer. In the example firmware the TxBuffer and RxBuffer size is limited to 10 characters as set in the source code. The firmware has 3 LEDs, a clock and UART peripheral as hardware that will be intercepted by HALucinator. The firmware will initialize and then ask the user to input 10 characters. It will then block until it receives the characters. When the rx buffer is full, the firmware will then print out the buffer and go into an infinite `do/while` loop.

Now that we know what the firmware will do, let us give some figures and examples of using HQ-Tracer. Figure 5.1 shows an example of what it looks like before launching the HQ-Tracer plugin. To run, the user will navigate to the script and press the green play button. Figure 5.2 shows the plugin asking the user to input the QEMU log file and shows the plugin docked right below the listing window in Ghidra. Figure 5.3 shows an example of what the plugin looks like with register state showing and the stack tracking enabled. In this case the execution starts in `Reset_Handler` and also goes to `__libc_init_array` and then to `main`. This figure shows the yellow and green highlighting of the program in the listing window (the left hand dock above the plugin), and in the decompile window, it has the current instruction highlighted in green. Figure 5.4 shows the end of trace, and you will see that the stack has the functions that it doesn't return to on the stack because of the infinite do/while loop.

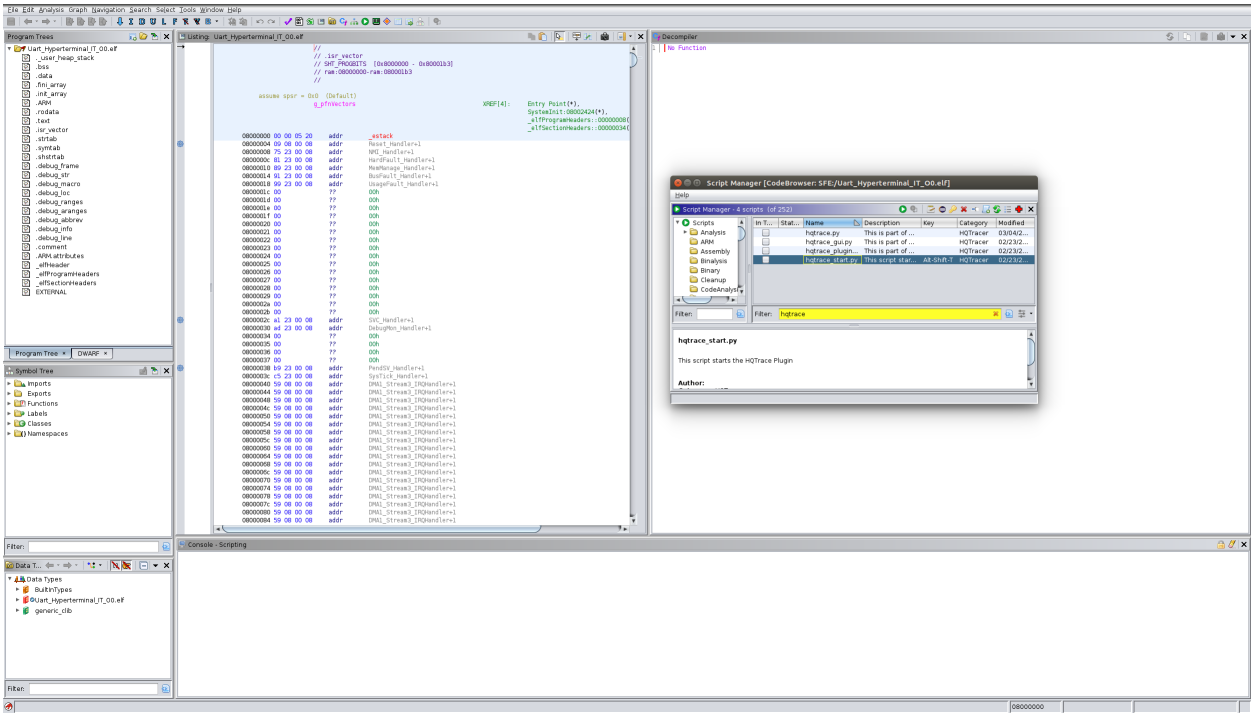


Figure 5.1. Example Ghidra window to start HQ-Tracer

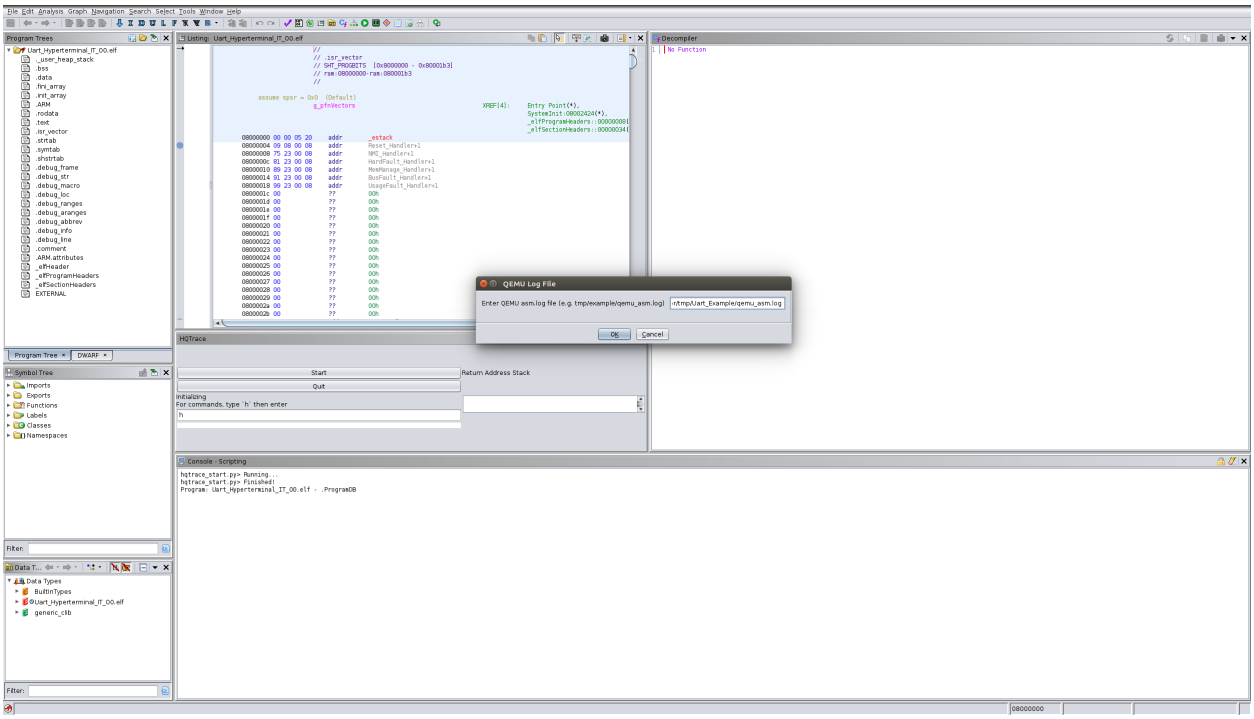


Figure 5.2. Example HQ-Tracer Prompt for entering QEMU asm log file

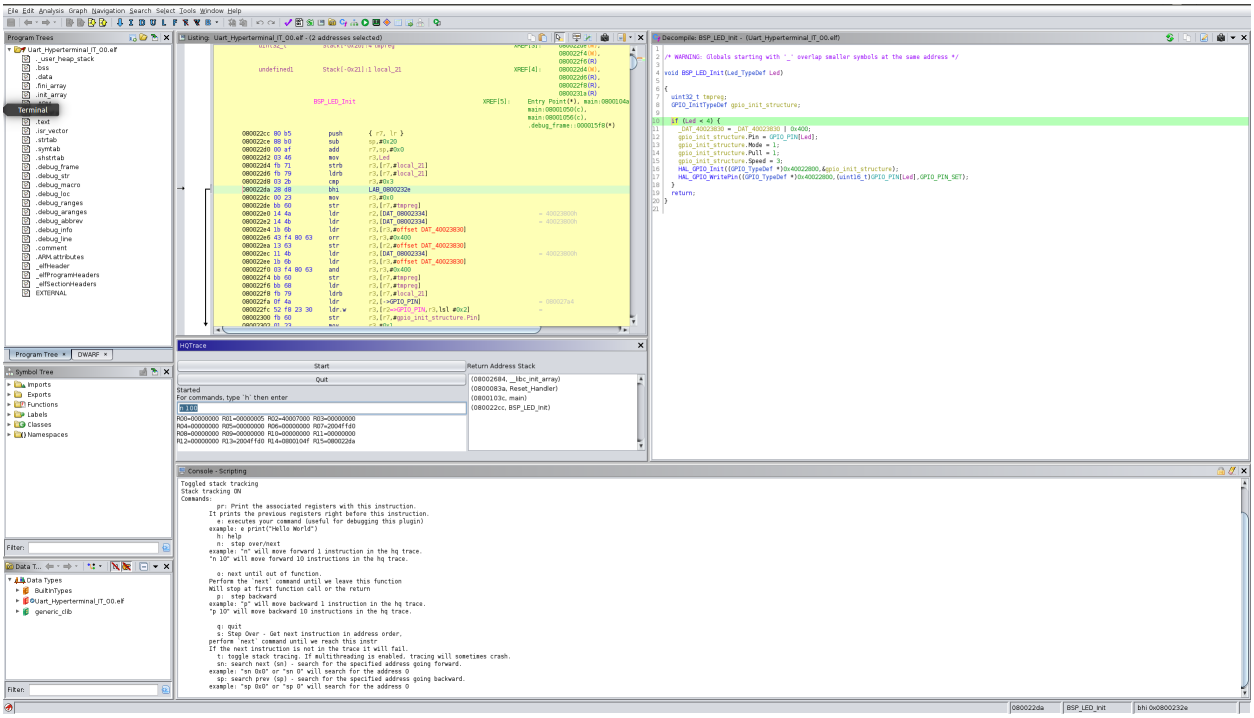


Figure 5.3. Example of HQ-Tracer Stack Tracing

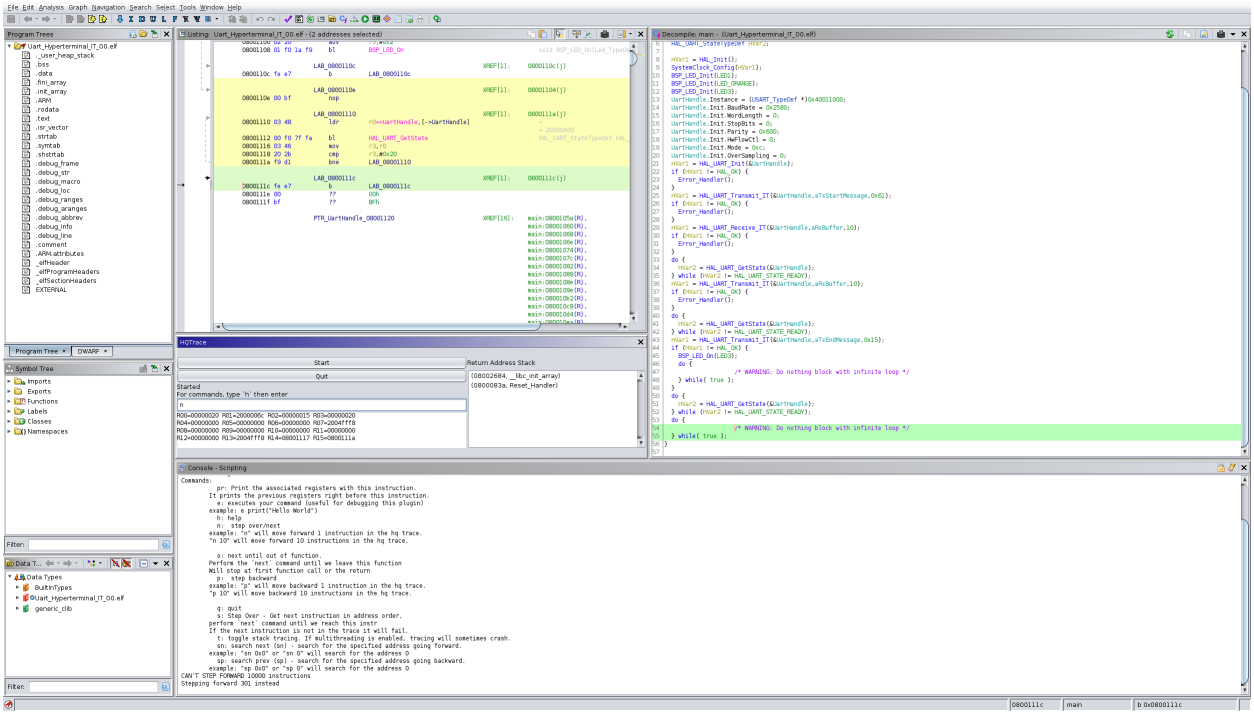


Figure 5.4. The end of an HQ-Tracer execution

6. PMATCH

This chapter discusses the need for function matching along with some of the approaches and their shortcomings and benefits. I introduce Pcode libMATCH (PMatch) and go through an example.

6.1 Motivation

Usually during RE we want to separate library functionality from the more interesting algorithmic code. In our case though, the library code is essential, as those are usually the functions that HALucinator will want to intercept. Because HALucinator will intercept hardware abstraction libraries, it is essential to actually know where to intercept, usually at hardware abstraction library functions.

It can be very difficult to understand what is happening in binary code, even when the disassembler and decompiler are working correctly. For example, in Figure 6.1 on the left hand side, we see what a partial flow graph would look like for the UART example from Section 5.3. Ghidra disassembles and decompiles correctly, but, as can be seen, it is a little difficult to determine what is actually happening. Contrast this with the right hand side of the figure, and we see that it is relatively easy to understand what is happening because of the useful function matching that has been performed.

Library or function matching is a difficult task, but is also one of the most useful tasks during reverse engineering. Understanding what occurs inside a binary is difficult, so when you have a binary/firmware that uses a library or functionality that you can match, it can be a huge time saver. A simple Hello World" program in C/C++ for example may have 1 `main` function, and 10 to >50 library functions in the compiled binary. By automatically separating and labeling the library functions, the algorithmic analysis, or library intercepting, can be performed rapidly without spending unnecessary time labeling. When matching functions, there are pros and cons for each technique. The most common ways to perform matching including binary code similarity analysis, hashing, and graph matching.

Because my prefered RE platform is Ghidra (for reasons mentioned previously), I wanted a plug and play tool integrated into my RE platform for function matching. I also wanted

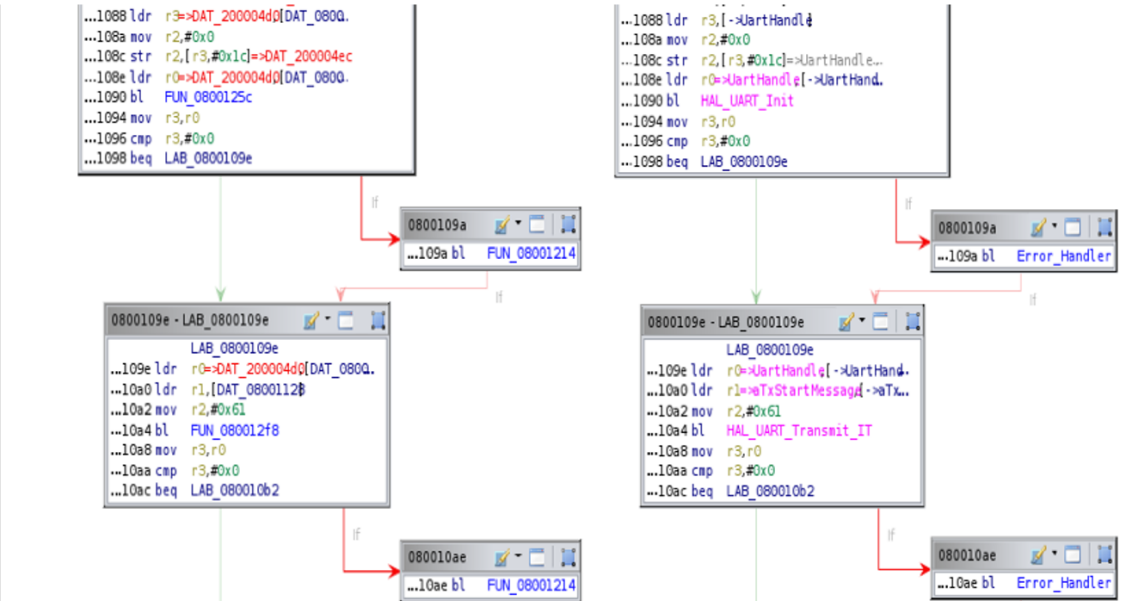


Figure 6.1. Example in Ghidra of a partial flow graph without function matching vs with function matching

a tool that could be fast for exact matches or that could allow for partial matching if there weren't exact matches. Before describing the implementation of PMatch, I elaborate slightly on popular matching techniques and mention the tools I have tried.

6.2 Approaches

Control Flow graph or Data Flow graph techniques will build a graph of the "database" functions, usually storing edges, nodes and other information such as strings or constants into the database. This can create large databases and take up a lot of memory quickly if the algorithm is not very efficient. After the graph database creation phase, during the matching phase, the tool/algorithm will attempt to solve the graph isomorphism problem (are 2 graphs equal). Depending on how large the functions are, and the extra data incorporated to the algorithm, this can be very slow.

Besides the graph matching technique, there is hashing. Hashing is probably the most popular way to do library matching. The technique is just as it sounds, taking a number of bytes from each function and hashing them. The number of bytes hashed, the techniques for storing, and the techniques used when there is a collision are the distinguishing factors between approaches using hashing. I will now briefly discuss the main hashing implementations in IDA and Ghidra, and then discuss the short comings of the libmatch tool that HALucinator was distrubuted with.

Hex-Rays created Fast Library Identification and Recognition Technology (F.L.I.R.T. or FLIRT) signatures [197] that will hash the bytes in each function in a library and create a signature database file. One of the goals of FLIRT is to reduce the amount of space needed when analyzing a binary to determine if there are library functions in it. Because of this, FLIRT will store bytes of the database functions in a compact tree structure for space savings and matching speed. One of the limits of this is that the tree structure is limited to 32 bytes, and if there are multiple functions that end up in the same leaf of the tree (all 32 bytes match), then they also store the CRC16 of the functions in the same leaf node from byte 33 until the first variant byte. This allows them to keep the database rather small. After you have FLIRT signatures and files loaded into your RE platform (in this case

IDA), during decompiling each byte is checked to see if it could be the start of a library function. It will match against the tree to see if there are any leaves in the signature file that matches, and if so will name the function. The downfall is that if different registers are used, or something minor is changed in the librar,y there will be no match. Another limitation is that short functions or functions without differentiable instructions are usually left out. This is usually ok, but it would be nice to be able to set the threshold manually for this.

Ghidra has what is called FIDB and Function ID matching. This is an extension built-in to Ghidra that is disabled by default. Ghidra's Function ID does something similar to FLIRT, hashing on the beginning bytes of each function. It is, however, relatively easy to create new database files with .fidb extensions to match against in Ghidra. While this technique is useful and we employ this on every firmware we RE, it is usually limited by access to compilers and source code, as for every signature file, is it necessary to have a compiled binary to create the database from. For proprietary compilers/tool-chains, such as VxWorks, this can severely limit this approach. To be the most successful, there needs to be a different signature file for each compiler and configuration, which is not always possible.

Hashing is closely related to binary code similarity where you compare two or more binaries against each other to see where there are similarities and differences. This approach is more useful when the user does not have the source code or access to a specific compiler toolchain, but they do have access to multiple binaries where at least one has functions labeled already.

HALucinator comes with a library matching tool [198] that uses angr, but it specifies that it is a preliminary tool and uses a *lot* of memory. This makes analyzing libraries with a large number of functions or a binary that has more than 50 functions difficult on a normal RE desktop/laptop. Beyond that, the tool was tested extensively for ARM Cortex+ architectures extensively but not other architectures. Other matching tools we have tried include Diaphora[199], REveal[200], among other binary diffing programs such as bsdiff[201], JojoDiff[202], HDiffPatch[203] and xdelta[204].

As a side note, some tool-chains make the function labeling process trivial, by embedding the symbol table into the binary, as is the case for VxWorks. However, if you are not so lucky as to have an embedded symbol table, reverse engineering one binary and reusing the work

for another binary (one that you suspect has the same tool-chain compilation and libraries included), would be ideal. This is where Pcode-libMATCH (PMatch) comes into play.

6.3 Implementation

PMatch was created with the goal of allowing matching between binaries or library/binary to allow the re-use of RE efforts. When using the emulation technique of Hardware Abstraction Layer replacement or High Level Emulation techniques in general, we want to specify addresses or symbols that we want to intercept and provide functionality stubs for.

PMatch is split into two different scripts. The first script (database script or DS) will be run on an existing binary that you have function names and symbols for functions of interest. The second script(matching script or MS) will be run on the binary you are analyzing that you do not have function names for.

The DS will cycle through every function and create a yaml database. This is not the most efficient way to store data, but it allows for tweaking the matching algorithm in the MS, rather than having to rebuild the database for matching. The DS will ask the user for the database file to save into. If this file doesn't exist, it will create a new file. It is important to note that it will append to the database without checking other entries in the database. This means there can be duplicates in the database, which can hurt the MS. The database saving algorithm will visit each function in the binary and save collected information to the database. Information that is gathered includes the main functions converted Pcode, all contants in the function, all strings in the function, the parameter count, the number of variables used, along with the pcode, contstants, and strings for any function that is called from within this main function. While I have found that not all of these are necessary for matching, it can be helpful when trying to differentiate various matches in the second script.

In the MS, the user is prompted for where the database YAML file is located, then they are asked for a filename where they want to save any resulting matches with addresses. While this is not strictly necessary, it is helpful when you want to test matches without actually renaming any functions in Ghidra. This MS algorithm will then read in the database and create a hash used for the actual matching. In this hash, the user can decide which things

from the database to include. They can also specify the minimum hash length, the max amount of time allowed for trying to match one function, and whether to allow a similarity matching. When similarity matching is enabled, the hashes are made into strings and an edit distance is used for matching the string hashes. This works because the hashes are number strings to begin with, so it will perform well if there are localized changes, but it will not perform as well when there are large reorganizations of code, such as basic block reordering or obfuscation.

In practice, this approach works well for exact matches, but depending on the metrics used for the database creation and matching, there is a wide range of success for similarity matching, with some binaries matching very well, and others performing at an unusable level. This is where tweaking the algorithm in the MS is useful, as changing a few things can make a big difference, especially when using similarity matching.

Another useful feature of the approach in Ghidra is that we can save memory by building the database file by file, not requiring the data to stay in memory. We can then create a database with a whole library by using the Ghidra headless analyzer. Some example commands of using this tools would be as follows (setting `$GHIDRA_HOME` to the path of the Ghidra installation). To first create the Pcode database for a given library (in this case for the cortexm3 library cross-compiled using arm-none-eabi-gcc) you would first run the DS (`pmatch_make_db.py`):

```
$GHIDRA_HOME/support/analyzeHeadless /home/someUser/someFolder PcodeDB
-scriptPath /home/someUser/PMatch/ -postScript pmatch_make_db.py
/home/someUser/ghidra_outputs/func_pcode_db.yaml -import
/home/someUser/PMatch/libmatch_tests/objects/arm-none-eabi/libmbed-cortexm3/*.o
-recursive
```

Then, for matching, you would open the binary you want to name functions in and run the `pmatch_match_funcs.py`. This will prompt you to enter a path for the yaml database, in our case it would be

```
/home/someUser/ghidra_outputs/func_pcode_db.yaml
```

You will then be prompted for the file of where you would like to save the matches (again in yaml format). This will save the matches into the file and if you would like to see how accurate the matching was you can then run `pmatch_match_funcs.py`. This will tell you what functions matched correctly and what functions matched incorrectly.

6.4 Example

Using the same example as in Chapter 5, I show a sample output from running the making database script and then matching against from the Ghidra GUI. This assumes you are creating the database from a single binary and matching against another binary. In this example, I use the .elf file to create the yaml database, then I match against a stripped elf and obj-dump into a binary format for matching against. In this way we know the ground truth and we also see the reliability that this technique has on the disassembler in Ghidra.

Figure 6.2 shows the start of the script for making the database and it shows the prompt asking for the full path of where to save the pcode database, including the extension. Figure 6.3 shows that we successfully finished running the database creation script and in this case there were 76 functions and their data saved to the database. Figure 6.4 shows the start of running the matching script, where it asks for the database filename and then an output filename of where to save the functions it matches on. Figure 6.5 shows the completion of matching and prints the number of functions that were in the database, how many non-conflicting hashes could be created.

In this example, only 68 of the 76 functions could have hashes created because the others conflicted. When manually checking the results, these functions had the exact same hashes and were short functions. PMatch also prints some of the statistics of the matching, and for this case we matched on 51 functions. Upon running a scoring script that takes in the ground truth and the output from this script, all 51 functions are correct matches, with 0 mistakes. However, you can see that this binary only has 61 functions (compared to 76 in the ground truth) that are recognized by Ghidra using the Auto Analysis, which is part of the reason more functions were not matched. When manually creating more functions in Ghidra at the spots that need functions, we can increase the matching to 63 functions out of the 76 in the

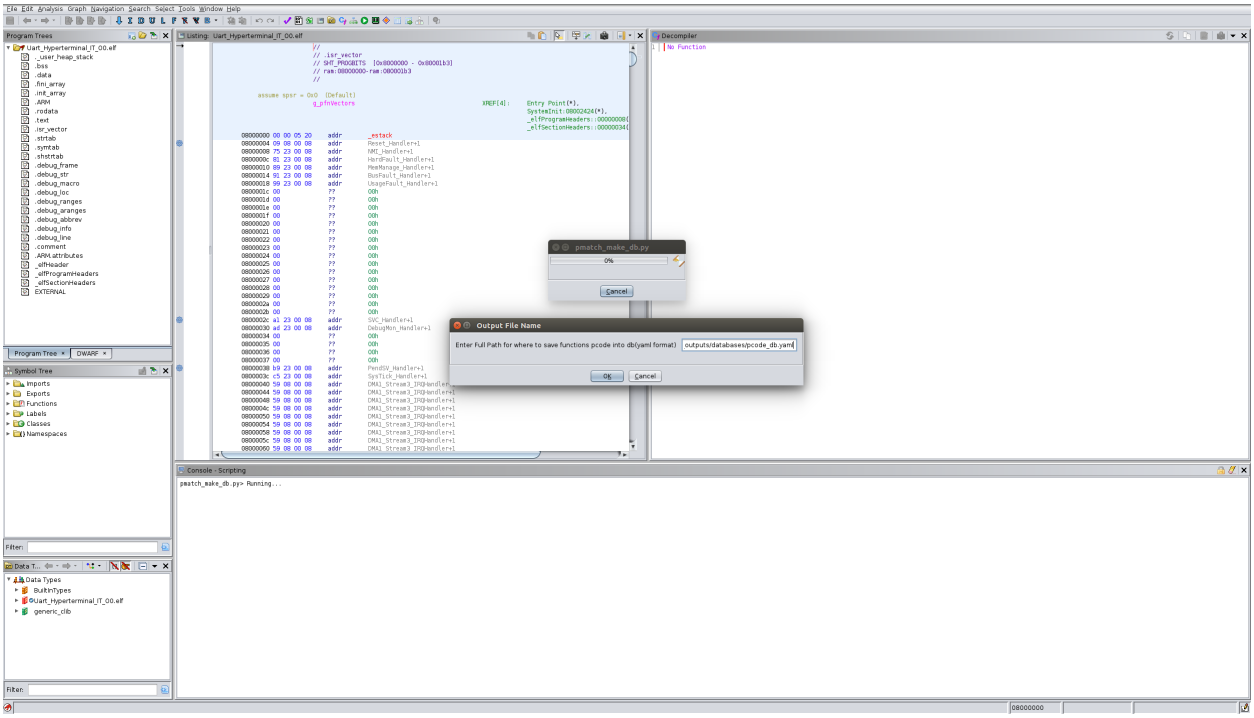


Figure 6.2. Example Ghidra window when prompting for the PMatch database creation

ground truth. Of these 13 non-matched functions, there are 8 that are discarded because of hash collisions.

In Table 6.1 you can see the addresses, ground truth function names and the names PMatch had a match on when using the default Ghidra analysis (not adding functions that Ghidra missed when comparing to the ground truth). As can be seen, there are no errors with the function names that were matched. This is not to imply that the tool is perfect or doesn't make errors, rather that with the right parameters and database, it can be useful in function matching.

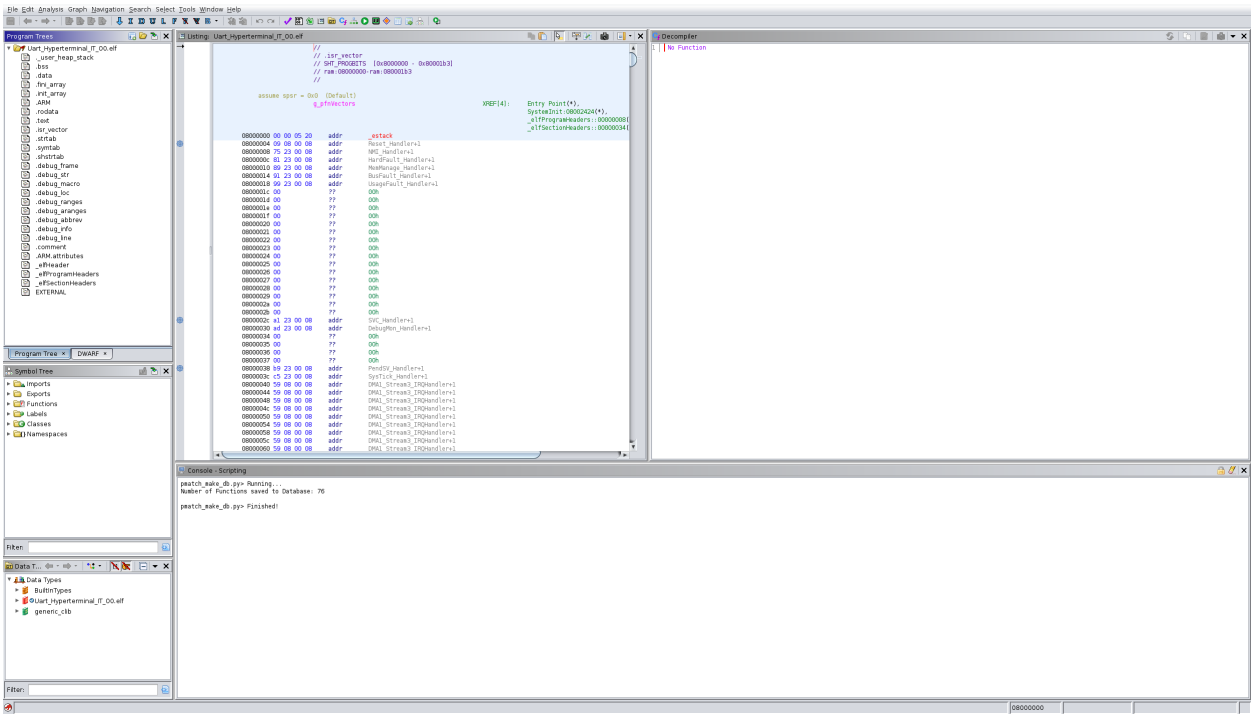


Figure 6.3. Example Ghidra window when PMatch database creation finishes

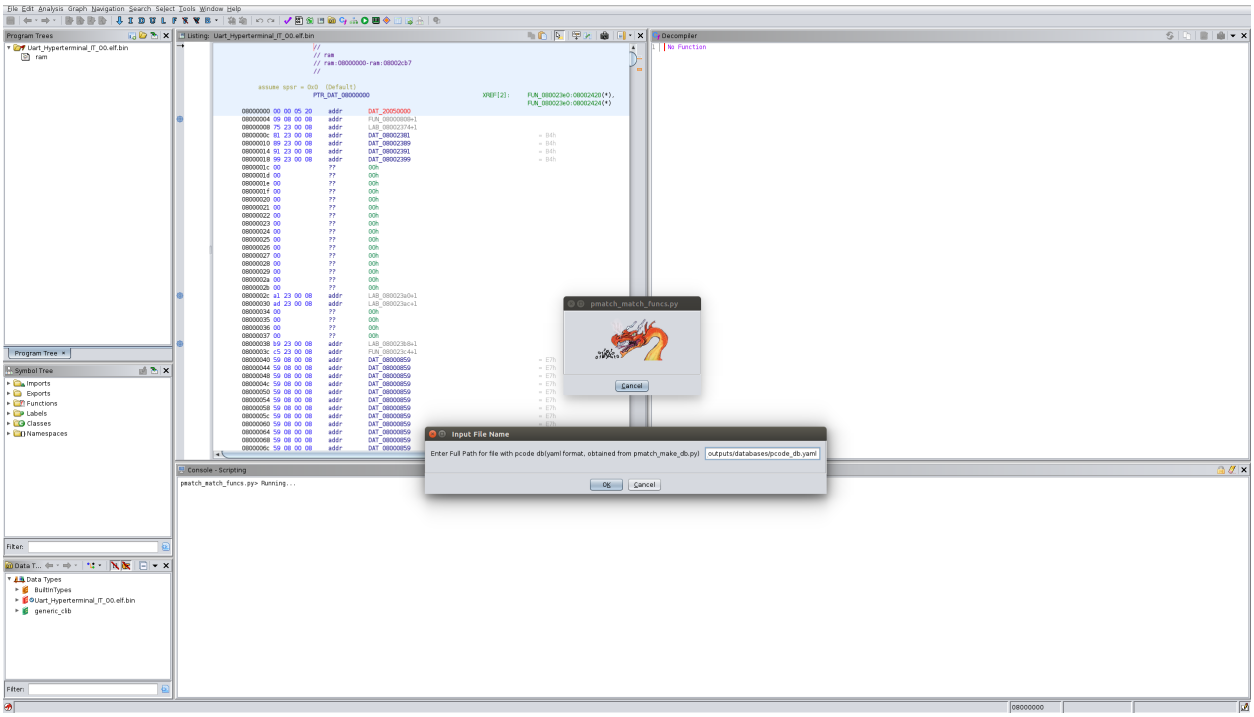


Figure 6.4. Example Ghidra window when prompting for the PMatch matching

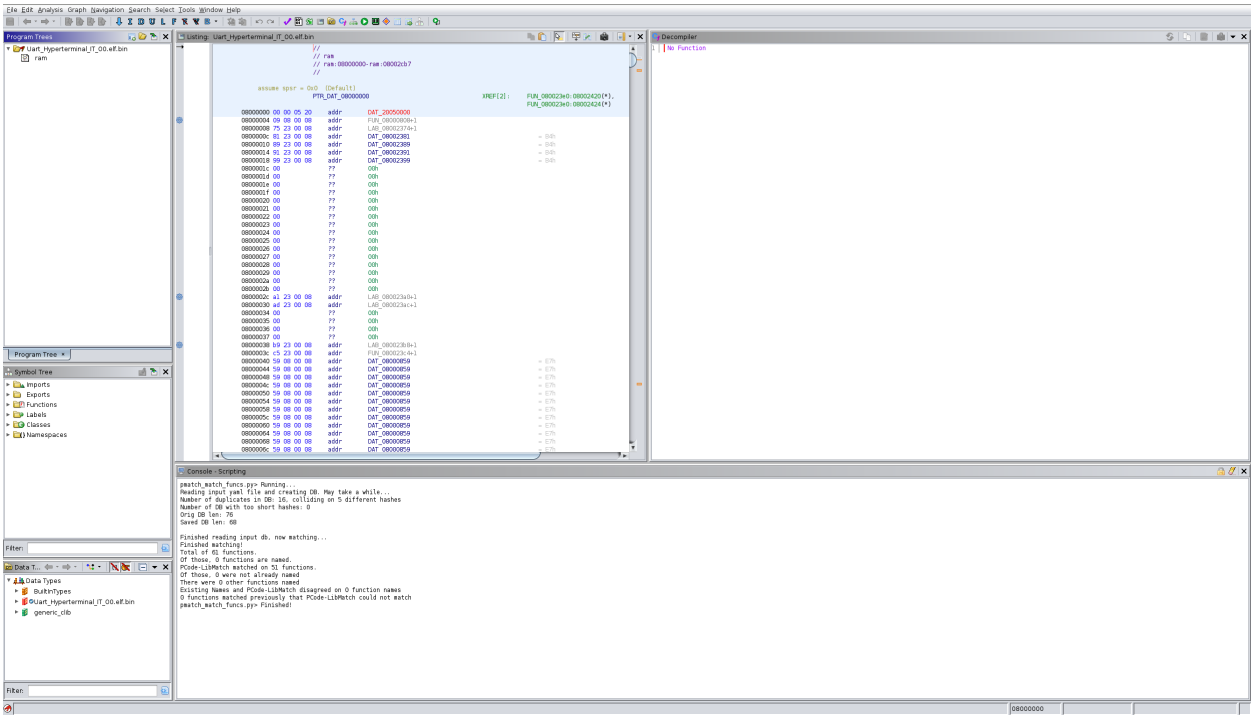


Figure 6.5. Example Ghidra window when PMatch matching finishes

Table 6.1. PMatch example matches with addresses

Address	GroundTruth	PMatchName
0x80001b5	deregister_tm_clones	
0x80001d5	register_tm_clones	
0x80001f9	__do_global_dtors_aux	__do_global_dtors_aux
0x8000221	frame_dummy	
0x8000259	__aeabi_uldivmod	
0x800025a	FUN_08000259	
0x8000289	__gnu_ldivmod_helper	__gnu_ldivmod_helper
0x80002b9	__gnu_uldivmod_helper	__gnu_uldivmod_helper
0x80002e9	__aeabi_idiv0	
0x80002ed	__divdi3	__divdi3
0x8000599	__udivdi3	__udivdi3
0x8000809	Reset_Handler	Reset_Handler
0x8000859	DMA1_Stream3_IRQHandler	
0x800085d	HAL_RCC_ClockConfig	HAL_RCC_ClockConfig
0x8000a21	HAL_RCC_GetSysClockFreq	HAL_RCC_GetSysClockFreq
0x8000b6d	HAL_RCC_GetHCLKFreq	
0x8000b85	HAL_RCC_GetPCLK1Freq	HAL_RCC_GetPCLK1Freq
0x8000bad	HAL_RCC_GetPCLK2Freq	HAL_RCC_GetPCLK2Freq
0x8000bd5	HAL_GPIO_Init	HAL_GPIO_Init
0x8000f21	HAL_GPIO_WritePin	HAL_GPIO_WritePin
0x8000f51	HAL_Init	HAL_Init
0x8000f95	HAL_MspInit	
0x8000fa1	HAL_InitTick	HAL_InitTick
0x8001001	HAL_IncTick	HAL_IncTick
0x8001025	HAL_GetTick	
0x800103d	main	main
0x8001135	SystemClock_Config	SystemClock_Config
0x8001215	Error_Handler	Error_Handler
0x8001221	HAL_UART_TxCpltCallback	HAL_UART_TxCpltCallback
0x8001235	HAL_UART_RxCpltCallback	HAL_UART_RxCpltCallback
0x8001249	HAL_UART_ErrorCallback	HAL_UART_ErrorCallback
0x800125d	HAL_UART_Init	HAL_UART_Init
0x80012f9	HAL_UART_Transmit_IT	HAL_UART_Transmit_IT
0x8001385	HAL_UART_Receive_IT	HAL_UART_Receive_IT
0x8001421	HAL_UART_IRQHandler	HAL_UART_IRQHandler
0x8001615	HAL_UART_GetState	HAL_UART_GetState
0x8001651	UART_EndRxTransfer	UART_EndRxTransfer
0x800168d	UART_DMAAbortOnError	UART_DMAAbortOnError

Address	GroundTruth	PMatchName
0x80016b5	UART_Transmit_IT	UART_Transmit_IT
0x8001761	UART_EndTransmit_IT	UART_EndTransmit_IT
0x8001791	UART_Receive_IT	UART_Receive_IT
0x8001885	UART_SetConfig	UART_SetConfig
0x8001cb9	HAL_UART_MspInit	HAL_UART_MspInit
0x8001d6d	HAL_DMA_Abort_IT	HAL_DMA_Abort_IT
0x8001db1	HAL_RCC_OscConfig	HAL_RCC_OscConfig
0x8002231	HAL_PWREx_EnableOverDrive	HAL_PWREx_EnableOverDrive
0x80022cd	BSP_LED_Init	BSP_LED_Init
0x8002341	BSP_LED_On	
0x8002375	NMI_Handler	
0x8002381	HardFault_Handler	
0x8002389	MemManage_Handler	
0x8002391	BusFault_Handler	
0x8002399	UsageFault_Handler	
0x80023a1	SVC_Handler	
0x80023ad	DebugMon_Handler	
0x80023b9	PendSV_Handler	
0x80023c5	SysTick_Handler	SysTick_Handler
0x80023d1	USART1_IRQHandler	USART1_IRQHandler
0x80023e1	SystemInit	SystemInit
0x800243d	NVIC_SetPriorityGrouping	NVIC_SetPriorityGrouping
0x8002485	NVIC_GetPriorityGrouping	NVIC_GetPriorityGrouping
0x80024a1	NVIC_EnableIRQ	NVIC_EnableIRQ
0x80024d1	NVIC_SetPriority	NVIC_SetPriority
0x8002525	NVIC_EncodePriority	NVIC_EncodePriority
0x8002589	SysTick_Config	SysTick_Config
0x80025cd	HAL_NVIC_SetPriorityGrouping	HAL_NVIC_SetPriorityGrouping
0x80025e1	HAL_NVIC_SetPriority	HAL_NVIC_SetPriority
0x8002619	HAL_NVIC_EnableIRQ	HAL_NVIC_EnableIRQ
0x8002635	HAL_SYSTICK_Config	
0x800264d	atexit	
0x8002659	__libc_fini_array	
0x8002685	__libc_init_array	
0x80026d5	__register_exitproc	
0x8002779	register_fini	
0x800278d	_init	_init
0x8002799	_fini	_fini

The address column shows the address in the binary. The Ground Truth column was obtained by using an .ELF file. The PMatchName column is the name that PMatch matched against in its database. An empty box means that PMatch did not match the function name.

7. GHALDRA

Ghidra HALucinator Emulator, or GHALdra (gawl-dra), is a Ghidra plugin for doing full or partial emulation. To understand why we made this tool and how it can be useful, I will first do another quick review of QEMU and HALucinator and compare.

QEMU is a quick emulator for multiple architectures. It is fast because it translates the firmware instructions to native host instructions as basic blocks, and then caches those blocks for faster emulation. With many applications performing computation in loops, this makes QEMU ideal for quick emulation. It also has support for a few peripherals along with the built-in support for many popular intruction set architectures. However, it is limited to the approximately 10 architectures that it supports. As anyone that has done hacking on QEMU can attest to, it is not always easy and can be a major pain to get things to work correctly. Adding new support and new devices is not trivial and can be a long painful process. This is where HALucinator can help tremendously. By using the technique of intercepting at certain breakpoints and providing stubbed functionality before changing the program counter, HALucinator allows for adding multiple peripherals without having to hack on QEMU. It is limited to the architectures QEMU supports, but for a large number of devices this is sufficient.

As alluded to above, the limits for both QEMU and HALucinator is the lack of ISAs that are supported. This is where Ghidra shines, as it has built-in support for >30 ISAs and there are more that are open sourced on Github. If however, the architecture you need support for by some chance does not exist in Ghidra, it is relatively straightforward to add support by providing a specification of how each instruction in your ISA translates into Ghidra's Pcode. After providing a few spec files, GHALdra and regular Ghidra emulation will work right away.

7.1 Ghidra Emulator

Ghidra has a built-in emulator in the `ghidra.pcode.emulate` class and functionality. In the Ghidra release you will find 2 example .c files under the docs and Emulation folder (docs/GhidraClass/ExerciseFiles/Emulation/Source) and 2 scripts that use the built-in emulator

to perform emulation. The DeobfuscateExample shows how to use some of Ghidra emulation functions to do dynamic analysis. The DeobfuscateHookExample puts a break point in the execution, showing how you can break and hook in the emulator. Using these examples and other examples online [45], [205], [206], we see that Ghidra has built-in writing memory, stack values, setting breakpoints, removing breakpoints and many other functionalities.

When searching for Ghidra emulators or tools, two tools stick out: TheRomanXpl0it user on Github with the repository ghidra-emu-fun [206] and the kc0bfv Github user and their repository pcode-emulator [45]. Each can emulate simple functions, but have different approaches behind them. TheRomanXpl0it uses the built-in Ghidra emulator whereas the kc0bfv gets the Pcode from Ghidra and then does execution of this Pcode itself. There is tremendous value in both approaches, with the emulation of Pcode standalone having an effect on external tools, with possible integration into those tools. In contrast, reusing the Ghidra Emulator will reuse many man hours of work from the NSA to get things working. The work I have done here has been concurrent with both of these other repositories, with my work starting on this emulator starting in May of 2019. The release of Ghidra was in March of 2019, and many users are currently working on emulation tools. I have implemented ideas from HALucinator, but these ideas have some support built-in to Ghidra and there is overlap between my emulator and these other emulators. The difference between HALucinator and my contributions are the way I have enabled partial emulation, allowing the user to start at various functions throughout a binary while still enabling intercepting and hooking of functions during emulation.

7.2 GHALdra Design

GHALdra used a skeleton framework from HQ-Tracer and integrated portions of both TheRomanEXpl0it and HALucinator into the framework. As such, GHALdra includes functionality from both of these tools in addition to new functionality. An overview of GHALdra design can be seen in Figure 7.1. The plugin is the primary of the whole design (controller). This plugin interacts with the emulator, the helpers, the intercept and callback master and the peripheral server.

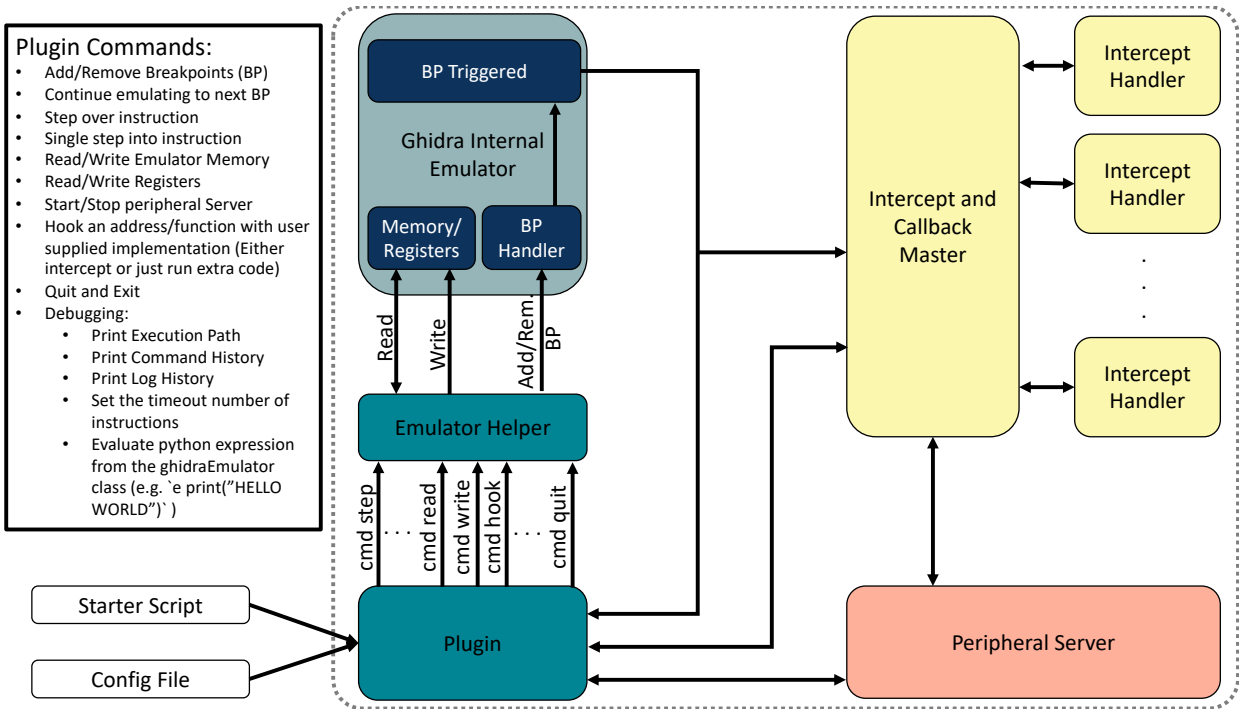


Figure 7.1. Design Overview for GHALdra

For the user to interact with the tool, there is a simplified interaction and a complex interaction available. The simple interaction includes starting the plugin and interacting with only the Ghidra emulator and emulator helper. If the user does not need to intercept or send information to/from the emulator to peripheral devices this interaction model works comprehensively. This type of interaction can be useful for reverse engineering software libraries inside firmware. In Figure 7.1, this interaction would include the boxes that are on the left hand side, colored in the teal and navy boxes.

An example usage for this type of GHALdra use would be if the user needed to RE a proprietary chip that does some sort of floating math operations but the chip you are RE does not include hardware support for floating point. In this case, most likely the operations are done with a software floating point library. When RE this type of situation, it can be helpful to emulate just the function, starting with various values and recording what mathematical operations are being done. In this way some functions can be named, such as add, subtract, etc.

For the complex interaction, it can be broken down in 2 sections, the Intercept Callback Master and the Peripheral Server. The simpler of these is when the Plugin interacts with the Intercept and Callback Master along with the Emulator and Helper. In Figure 7.1 the boxes included in this type of interaction will be the teal and navy boxes as before, but now it will include the yellow boxes as well. In this case, the user will intercept at certain addresses/functions and provide functionality of what to do, usually with a Python function.

For example, if there is an external `puts` function or `printf` function, this is where the user could provide their own implementation of that functionality in Python and hook at that function. The user will also specify if they want to *intercept* the function, or if they want to just insert added code for functionality/book keeping. An example of where the user would not want to intercept the function is when they want to log or print a function for a subsequent frequency execution analysis of the function. This would be accomplished by hooking the function/address but not intercepting, as intercepting will change the normal execution flow of the binary.

In the most complex interaction model, the Plugin will interact with Emulator, Helper, Intercept and Callback Master, and the Peripheral Server. In Figure 7.1 this includes all of

the boxes in the figure. The Plugin will start the Emulator and the Helper, register intercepts and callbacks with the Intercept and Callback Master, and it will also start the Peripheral Server. In turn, the Peripheral Server will also interact with the Intercept and Callback Master. The Intercept and Callback Master will also be triggered by any break point in the Emulator. This is useful when performing Emulation with the need to allow asynchronous external events to influence execution, such as when receiving UART communication via an interrupt, where the user will provide intercepts handlers for peripherals and anytime the Emulator, or Intercepts, try to interact with that hardware, it will communicate with the Peripheral Server and the models/handlers at that level. An example of this type of interaction is detailed in Section 7.4.

7.3 GHALdra Implementation

The base of GHALdra is similar to the HQ-Tracer Plugin. The GUI has handlers to send actions to the Plugin when the 'Start' and 'Quit' buttons are pressed, and it also has a message label that tells the user whether the plugin has been started or not. The register state is shown below the comand input box. This register state is updated every time the emulator is stopped or every 100 instructions, whichever is less, as this allows the user to see the emulator is indeed working. The command input works in the same way that the HQ-Tracer worked, by sending the entire input to the Plugin.

The Plugin will interact between the GUI and the main GHALdra class and action algorithms. The plugin will take commands from the GUI and if they are emulating commands will start those commands in a thread, setting the thread as a daemon. This allows the user to send other commands to the plugin even if it is busy executing instructions in the emulator. This allows for breaking the execution without having to close Ghidra, as the monitor for plugins does not pop up in the same way as an execution script. If the command sent is not an emulation command then the plugin will send the command to execute with the main logic in the main thread. As the non-emulating commands are quick to complete, this ensures less opportunity to crash the plugin, even with rapid command entry. I also enabled fuzzing of commands and data from TheRomanExpl0it repository. This enables fuzzing the

inputs to the function that you want to emulate in Ghidra. This means that the input bytes are randomly generated from the available ranges specified by the user, and then emulation is started. All additions are noted in the software when any other code was used, and the license was checked and verified to ensure using the code would be in compliance.

Part of design decision when creating GHALdra was to enable reuse of as many handlers as possible from HALucinator. HALucinator has handlers that work for Atmel ASF v3, MBed, STM32f4, and some VxWorks operating system handlers. These handlers include functionality for ethernet, bluetooth, rf233, certain sd cards, timers, uarts, usarts, serial ports, spi, wifi, and other gpio. These do not work for every architecture, but the reuse value in them is very high. In recent work in collaboration with Sandia National Laboratories we have used HALucinator to partially emulate VxWorks OS based devices. These are real world devices including a Schneider Electric SCADAPack 350 remote terminal unit, a Schneider Electric Modicon 34- programmable logic controller, and a Hughes 9201 BGAN inmarsat terminal.

To be able to reuse the work on these devices, the VxWorks handlers and the other handlers for GHALdra, part of the design decision included making the design compatible with HALucinator. To accomplish this I implemented a master intercepter that works similar to how Avatar^2 registers breakpoints for QEMU. This master class will do all of the intercepting and registering at the address we want to hook at, along with what class to call when the breakpoint is actually triggered. This will interact with the break point handler class that does the registering and the individual intercepts. When the breakpoint is triggered, the address will be looked up in the master interceptor before calling the relevant implementation. If the handler returns `True`, then the value returned along with the boolean will be saved in the return location and the function will be skipped. We then skip the function by attempting to go the next address in address-order from where the function was called. We rely on getting this information from Ghidra, and if there is an error during emulation, it is expected the user will manually fix the binary or fix the error by setting the program counter manually in the handler.

Because HALucinator allows for intercepting an address or a function, I also implemented an address interceptor that will not change the program counter or try to save return infor-

mation to the return location (depending on the language specification). This functionality is similar to the function interceptor, but it allows the user to specify in the configuration file whether each intercept is an address intercept or not. This address interceptor functionality is also the backup implementation if the function interceptor has an issue during hooking, as the address interceptor has less logic where it can fail.

After the basic interceptors were working, I branched out to more complex handers, where the peripheral server is required. This required implementing the peripheral server in a way that works with HALucinator. HALucinator uses the Python ZMQ implementation to send messages between the handlers and the peripheral server. This is a problem because the Python ZMQ library uses C compiled libraries, and hence will not work with Jython. To work around this, we need to integrate the Java implementation of ZMQ (JeroMQ [207]) into Ghidra and slightly change the handlers API. To accomplish this in Ghidra version <= 9.1, it is possible to download the JeroMQ, use Maven to build it and then add the `target/classes/` folder to the `LD_LIBRARY_PATH`. Another option would be to instead add the JeroMQ .jar to the plugins folder instead. This did not work for me in Ghidra >= 9.2, rather we have to build with maven, then copy the `target/jeromq-0.5.3-SNAPSHOT.jar` into the `Ghidra/patch/` folder. This should work if you are able to download just the .jar file for your architecture as well, it is not necessary to build the .jar yourself. This is also how a user can use other Java libraries that are not included in Ghidra without having to build Ghidra from scratch.

As a side note, to install Python libraries such as NetworkX or PyYaml, it is possible to use the built-in Jython package manager. If the library has any C compiled portions, this will usually not work. However, many packages will have a pure Python implementation that is a little slower than the C compiler version, but it will work with a special compile command. To enable shorter examples, I will show a few commands that I will use so that things are setup properly in the environment. I assume that `$GHIDRA_HOME` has been exported correctly and is in your startup script.

```
`alias jython='java -Dpython.cachedir.skip=false -jar
$GHIDRA_HOME/Ghidra/Features/Python/lib/jython-standalone-2.7.2.jar`
`jython -m ensurepip`
```

```
`rm $GHIDRA_HOME/Ghidra/Features/Python/data/jython-2.7.2/Lib/site-packages -r`
`ln -s $GHIDRA_HOME/Ghidra/Features/Python/lib/Lib/site-packages
$GHIDRA_HOME/Ghidra/Features/Python/data/jython-2.7.2/Lib/site-packages`
```

Now that this has been done (only one time per Ghidra release) I can show a short example of how to install PyYaml. These commands are relatively straightforward:

```
`git clone https://github.com/yaml/pyyaml.git`
`cd pyyaml`
`jython setup.py --without-libyaml install`
```

For simpler packages where you don't need to use an extra flag you should be able to install using:

```
`jython -m pip install networkx=2.2`
```

7.3.1 GHALdra Options

The command line options that have special handlers are shown in Table 7.1. These are the built-in forward facing commands, but there are other commands that can be executed by using the `e` command that is forward facing. These commands include loading an HQ-Trace and playing with the address set for coloring in Ghidra, changing memory and printing debugging support messages. Interesting commands for changing execution of the binary include the `hook`, `wm`, `wr`. These commands allow the user to dynamic change the memory or registers as they are emulating the binary.

7.4 Example

I will use the running STM32 UART example from Chapter 5. In Listing 7.1 the config file is shown. As can be seen, the UART and other Hardware Abstraction Libraries are intercepted. In most cases all we need to do is return a 0 or skip the function. In the case of the trasmit and receive for the UART, we will actually intercept them. The handlers will interact with the peripheral server, sending ZMQ messages between the emulator and the external device. For the full example to work like the actual hardware, we will need to open

Table 7.1. GHALdra Command Options

Command	Explanation
b	'b 0xXXXXXXXX' - add breakpoint ('hex_address')
prukraintpath	print the last X instructions executed (x is specified as numInstrToSave in config file)
c	continue - execute up to sys.maxint or timeout number of instructions in program
read	read memory addr (either 'hex_from:hex_to' or 'hex_from size')
d	d: 'd 0xXXXXXXXX' - remove breakpoint ('hex_address')
break	break: break at the current instruction. Can be used to jump out of infinite loop
e	executes your command. e.g. 'e print("Hello World")'
h	help - prints all the available commands
l	prints a serialized version of this debugging session
n	n: 'n [x]' - This will step over x instructions. It will count any function call as a single instruction, executing the whole function
timeout	timeout: 'timeout x' Set number of instructions to timeout on to x. e.g. 'timeout 10' followed by c will execute 10 instructions
p	print state
q	Quit
s	's [x]' - This will execute [x] instructions, even if it steps into a function. If x is not specified it defaults to 1
hook	'hook address module.function' Replaces a function with a python implementation e.g. hook 0x080355d8 halucinator.bp_handlers.SkipFunc
wm	'wm hex_addr hex_bytes' - write memory addr e.g. wm 0x0 1a2b
wr	'wr reg_name value' - write a register e.g. 'wr r2 2960982560' or 'wr pc 0x01234')

a separate terminal and run the UART external device. When we do this it will act as if it is the other device we are communicating with in the firmware. The example command for this case would be `jython halucinator/external_devices/uart.py -i=1048896` (the jython in this case is an alias for the stand-alone jython.jar is located in Ghidra).

In Figure 7.2 we see the typical Ghidra window before any plugins are activated. To start GHALdra, type in `ghaldra` and select the `ghaldra_start.py` script before pressing the play/run button. During initialization, GHALdra will ask the user for a configuration file, which in this case is shown in Listing 7.1. This will specify the entry point for where

Listing 7.1GHALdra Config Example

```
logfilename: /home/someUser/ghidra_outputs/ghidra-emu.log
output_directory: /home/someUser/ghidra_outputs/
debug: False
numInstrToSave: 10
entry_point: 0x0800103c
breakpoints:
  - 0x0
intercepts:
# ------------------------------UART ---------------------------------------
- class: halucinator.bp_handlers.stm32f4.stm32f4_uart.STM32F4UART
  function: HAL_UART_Init
  addr: 0x0800125c
- class: halucinator.bp_handlers.stm32f4.stm32f4_uart.STM32F4UART
  function: HAL_UART_GetState
  addr: 0x08001614
- class: halucinator.bp_handlers.stm32f4.stm32f4_uart.STM32F4UART
  function: HAL_UART_Transmit_IT
  addr: 0x080012f8
- class: halucinator.bp_handlers.stm32f4.stm32f4_uart.STM32F4UART
  function: HAL_UART_Receive_IT
  addr: 0x08001384
# # ------------------------------Generic ----------------------------------
- class: halucinator.bp_handlers.generic.timer.Timer
  function: HAL_GetTick
  addr: 0x08001024
- class: halucinator.bp_handlers.ReturnZero
  function: HAL_Init
  addr: 0x08000f50
- class: halucinator.bp_handlers.ReturnZero
  function: HAL_InitTick
  addr: 0x08000fa0
- class: halucinator.bp_handlers.Counter
  function: HAL_IncTick
  addr: 0x08001000
- class: halucinator.bp_handlers.ReturnZero
  function: HAL_MspInit
  addr: 0x08000f94
- class: halucinator.bp_handlers.ReturnZero
  function: HAL_RCC_ClockConfig
  addr: 0x0800085c
- class: halucinator.bp_handlers.ReturnZero
  function: HAL_RCC_GetHCLKFreq
  addr: 0x08000b6c
- class: halucinator.bp_handlers.ReturnZero
  function: HAL_RCC_GetPCLK1Freq
  addr: 0x08000b84
- class: halucinator.bp_handlers.ReturnZero
  function: HAL_RCC_GetSysClockFreq
  addr: 0x08000a20
- class: halucinator.bp_handlers.ReturnZero
  function: HAL_RCC_OscConfig
  addr: 0x08001db0
- class: halucinator.bp_handlers.ReturnZero
  function: HAL_SYSTICK_Config
  addr: 0x08002634
- class: halucinator.bp_handlers.ReturnZero
  function: HAL_PWREx_EnableOverDrive
  addr: 0x08002230
```

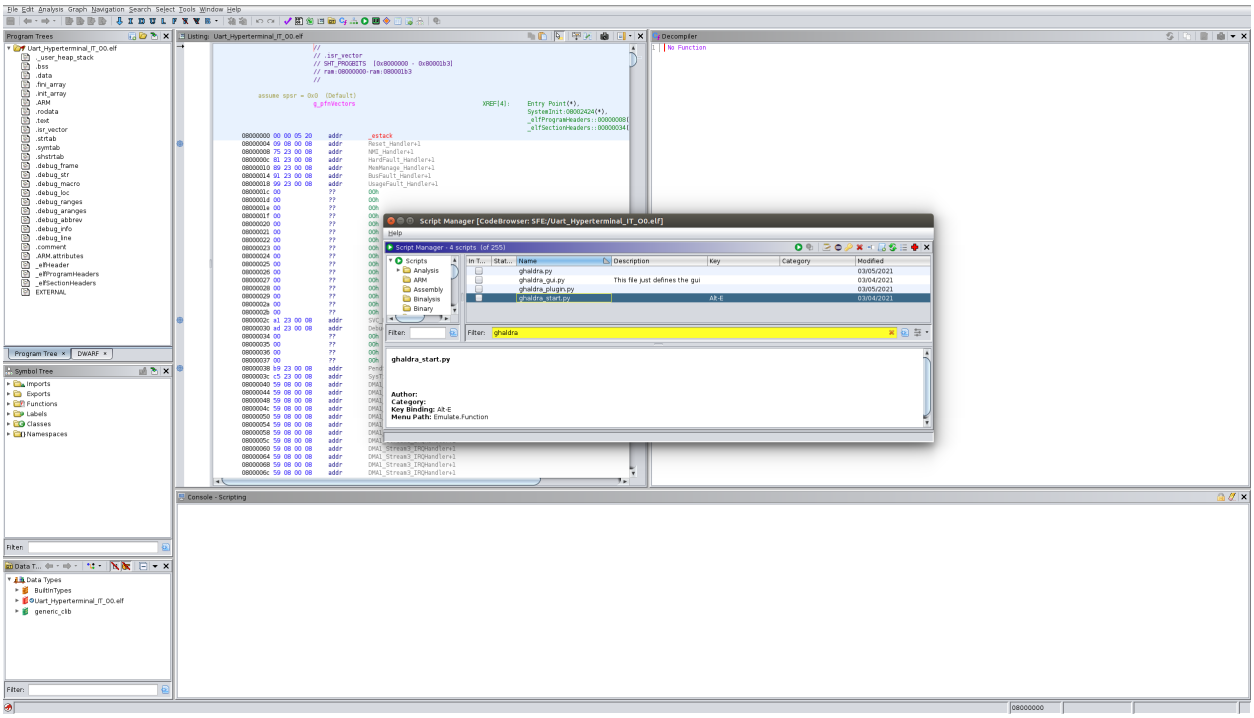


Figure 7.2. Example Ghidra window to start GHALdra

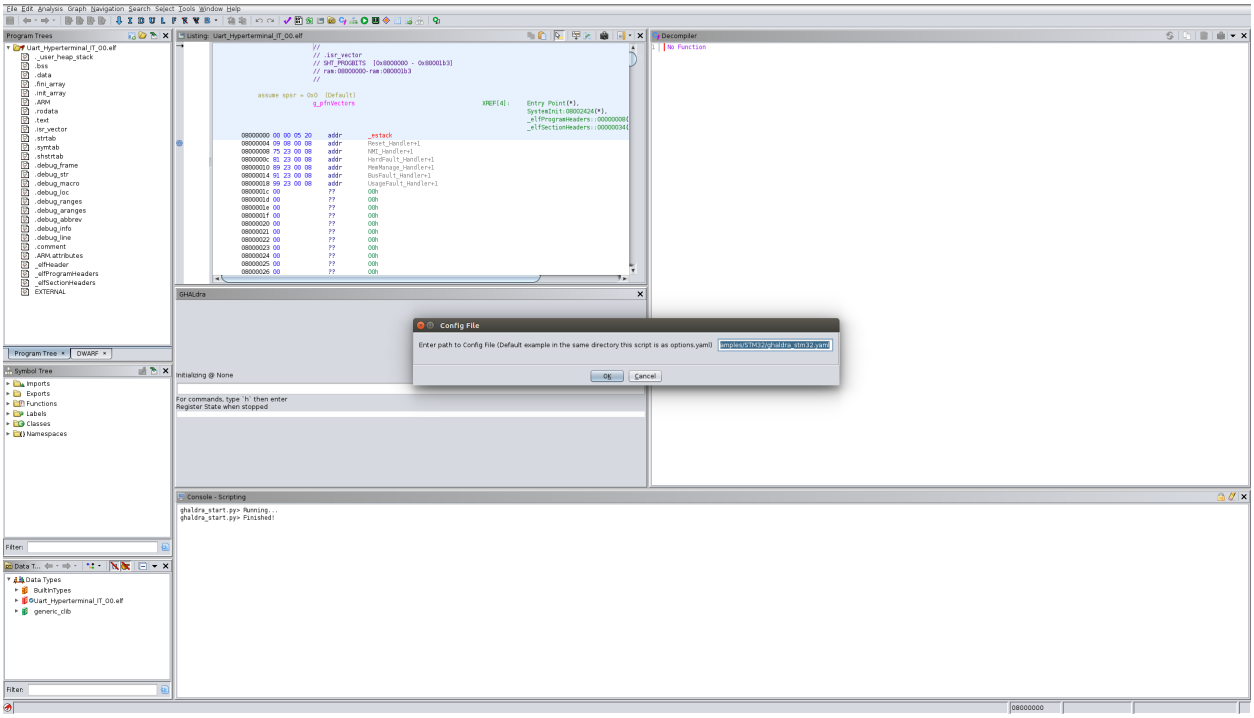


Figure 7.3. Example Ghidra window during initialization of GHALdra

we want to start emulating, a few output options, as well as any breakpoints and intercepts. GHALdra will read this information in, set the appropriate settings and attempt to hook all of the intercepts and set all of the breakpoints. As seen in Figure 7.4, after initialization, GHALdra will start the emulator and jump to the entry point. The registers are not shown at this point because no instructions have been executed. As soon as a single instruction has been executed, the registers will show up in the box below the command entry as seen in Figure 7.5. We can then use the 'continue' command `c` and the emulation will continue. Seen in Figure 7.6, at this point the emulator is blocking, waiting for input into the RX buffer. On the terminal that is acting as our peripheral device we see the message that the emulator sent over the UART through the peripheral server Figure 7.7. It asks us to input 10 characters using the keyboard. At this point we type in some characters and press enter. If we don't enter 10 characters the emulator will still block until the buffer has 10 characters. In this example I entered more than 10 characters, as only the first 10 are put into the buffer. We press enter and at this point the emulator will start running again, and in our TX handler we print what we put into the buffer, so it shows only the 10 characters in

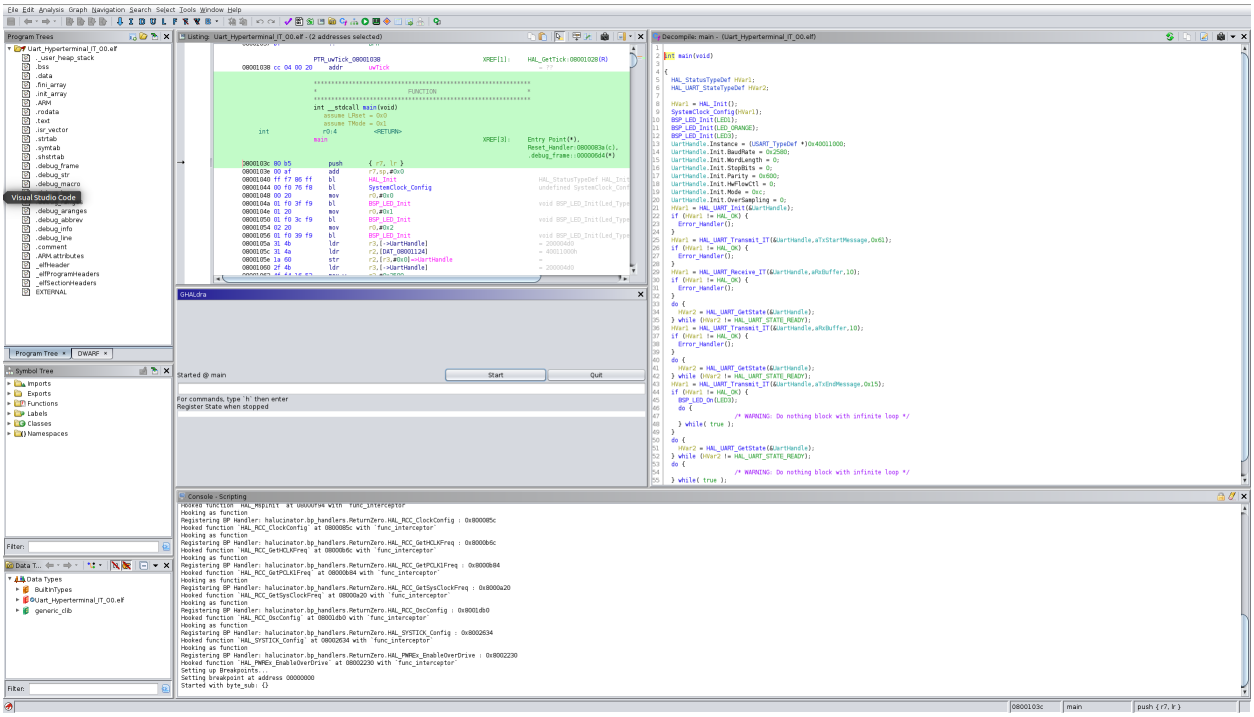


Figure 7.4. Example Ghidra window after configuration is loaded GHALdra

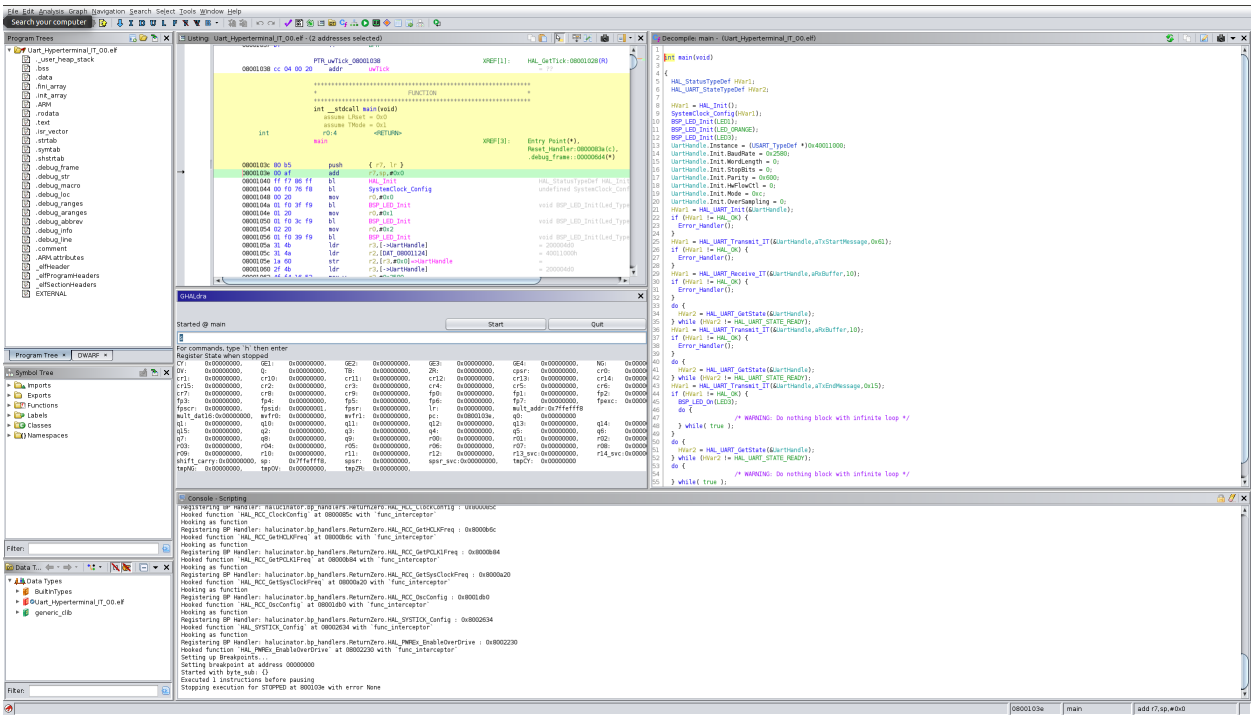


Figure 7.5. Example Ghidra window after doing a single-step in GHALdra

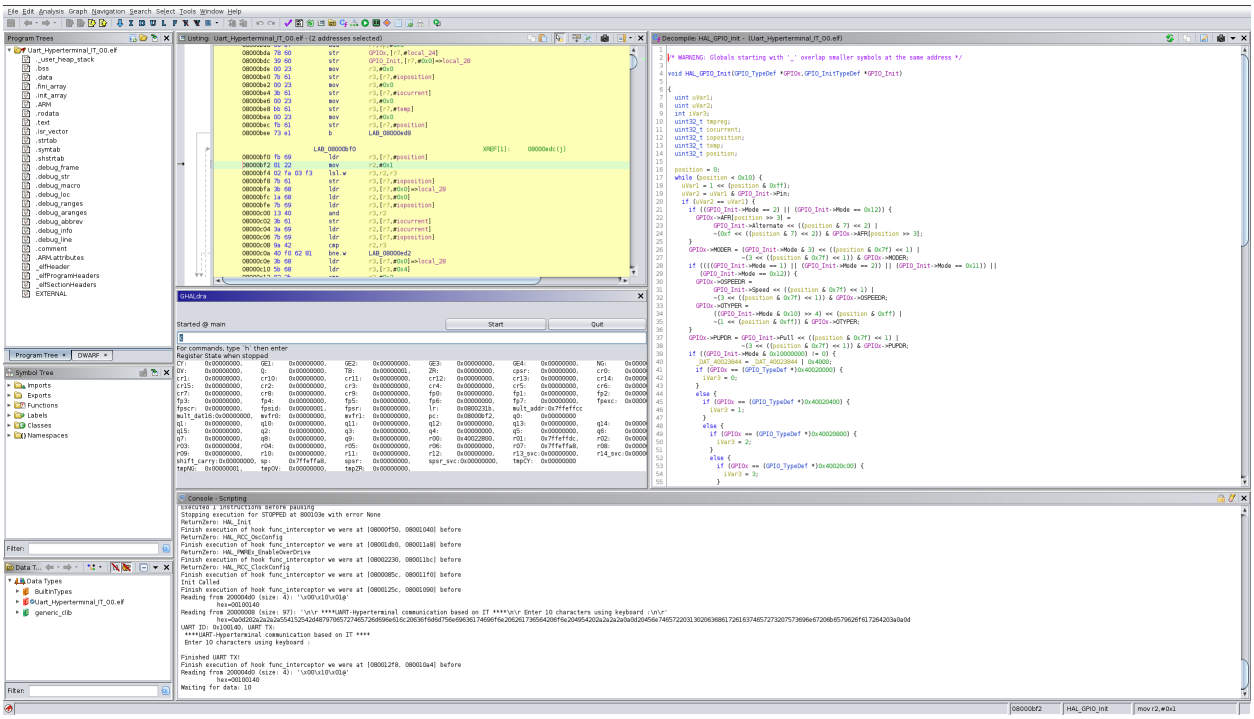


Figure 7.6. Example Ghidra window after continuing in GHALdra

```
(sfe) chrwrig@ubuntu1604:~/sfe/GHALdra$ jython halucinator/external_devices/uart.py -i=1048896

 ****UART-Hyperterminal communication based on IT ****
 Enter 10 characters using keyboard :
```

Figure 7.7. Example Terminal window running UART external device communicating with GHALdra through the peripheral server

```
(sfe) chrwrig@ubuntu1604:~/sfe/GHALdra$ jython halucinator/external_devices/uart.py -i=1048896

 ****UART-Hyperterminal communication based on IT ****
 Enter 10 characters using keyboard :

GHALdra Rocks!
GHALdra Ro

 Example Finished
```

Figure 7.8. Example Terminal window running UART external device after we enter in our characters and press 'Enter' which sends the data through the peripheral server to GHALdra

the buffer Figure 7.8. Back in the Ghidra window we can then see that we get to an infinite loop Figure 7.9, which for embedded firmware is essentially the exit point if we are not using interrupts.

7.5 Interrupts

In many firmware applications that are simple, we do not care about intercepts as they do not utilize them. So, supporting intercepts may not be necessary depending on your reasons for emulation. If, for example, you just want to see the software functionality of certain functions, interrupts are not necessary. However, if you have multiple tasks that you want to see how they interact, you may need the interrupts. I will mention that GHALdra is probably not the most efficient for this use case, as Ghidra Emulator in general does not do the caching like QEMU does for faster emulation. GHALdra is meant to be seen in real time, meaning slower execution and not for timing analysis.

With this disclaimer, GHALdra has support for interrupts. To be compatible with HALucinator we have handlers that act like QEMU does with qmp. It will allow you to send interrupts and stop emulation with the IRQ or FIQ, and it will save the current register context, but it does not know where the handlers are located. The user will have to provide the handlers, or locations in the firmware for the handlers, before the interrupts will work correctly.

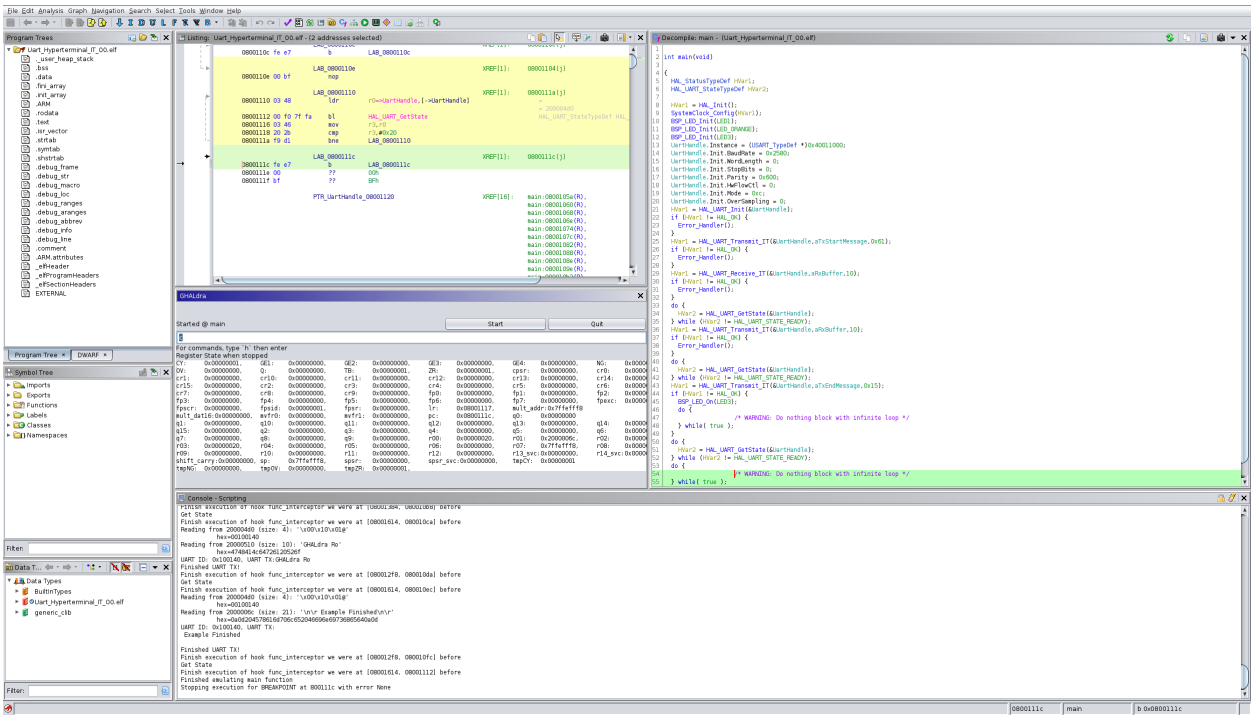


Figure 7.9. Example Ghidra window when GHALdra has finished re-hosting the UART firmware

While GHAldra can do many things, I do not see it as a kill-all, rather as a tool that can be used to help RE, find vulnerabilities, help understand a firmware, and even help create a specialized emulator when needing timing or interrupt analysis. Depending on the hardware we run Ghidra on, and the speed of the system we are emulating, GHALdra may be a sufficient solution, but that is not its primary purpose. The primary purpose is to emulate parts of firmware such as a single task, like a terminal application. For example, VxWorks embeds a terminal into firmware. This terminal may be disabled by the firmware, but using this tool we can jump to the terminal and interact with the firmware to understand what is happening and what the functionality is in the firmware.

8. ADDITIONAL SOLUTIONS

GHALdra is a wonderful solution for many challenges encountered during firmware rehosting, but it does not have tools built-in to address all challenges. To this end, I include other reverse engineering tools and scripts that I have created that integrate into Ghidra so that the platform can be as comprehensive as possible when addressing challenges.

While the newest release of Ghidra (v9.2.2) has some graphing capabilities, I have found them to be lacking in certain areas. Before a graphing service was included in Ghidra, I developed scripts that use Python NetworkX library to do graphing of control flow and data flow. This is similar to Ghidra's built-in graphing, using *varnodes* in Ghidra as nodes in the graph for data flow analysis and instructions for control flow analysis. NetworkX is already integrated with various printing and visualizing techniques included the universal .dot graph formats that can be imported to many graph visualization tools.

In Ghidra, writing scripts to do various static analysis tasks is relatively straightforward, even if there is not a lot of documentation outside of doxygen for certain classes. This enabled writing loop analysis scripts to search for infinite loops inside a firmware (as infinite loops are usually a bug, rely on an interrupt, or expect interaction from hardware). In these scripts, it is useful to create bookmarks for quick navigation to loop, and it is also useful when scrolling through code to see the colored loops quickly in the disassembler window.

Searching is a common task in static analysis and reverse engineering. I have created scripts that attempt to find certain functions of interest, including context switches, entrypoint, printf and other library functions. This is accomplished by specifying patterns and searching through Ghidra functions. For example, to search for a context switch in an ARM binary you will usually look for most of the registers to be saved along with the stack pointer, link register, and pc (for ARM). By searching for functions that either save to, or save from, all of these registers greatly reduces the number of functions that we need to analyze to find the context switch. Using this same kind of technique, I have written an escapes analysis which will try to determine the variables inside functions that access global functions/variables or stack variables. By doing something like this we could determine

whether a function is safe to skip or return a constant from when performing High Level Emulation using intercepts and hooking.

When analyzing a binary, it is also necessary to determine the instruction set architecture, the base address, and the entry point. I have provided brute force scripts that import a given binary into Ghidra for all possible architectures, does a static auto analysis, and saves the stats into a yaml file for digesting by the user. This can help narrow down the architecture a given binary was created for. Beyond the architecture script I have also included a script that attempts to guess the base address of the firmware by doing a strings and dword comparison.

While these scripts are not encompassing, or complete by any stretch of the imagination, they have been useful when trying to RE and emulate real world binaries.

9. VXWORKS

This chapter focuses on how HALucinator and GHALdra was extended to work with VxWorks by creating a Re-hosting Support Layer. This work was first done on HALucinator in collaboration with Sandia National Laboratories and a modified part of this chapter was published at Binary Analysis Research (BAR) an NDSS workshop in 2021 [26]. Challenges that were encountered this chapter are discussed throughout the thesis and specifically in [208]. In this chapter I will discuss how this is done with regards to HALucinator and GHALdra interchangeably, as the GHALdra design was implemented so that it would work and integrate with HALucinator.

VxWorks is a popular Real Time Operating System (RTOS) from Wind River, that is estimated to be run on over 2 billion devices [209] and has been on embedded systems for over 30 years, after starting out as VRTX [210]. It is commonly found in safety-critical industries such as aerospace, automotive, medical, and manufacturing [211]–[213]. It is used extensively by major companies including Siemens, Boeing, Bosch, Huawei, Northrop Grumman, and others. 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 [214]. VxWorks has also been demonstrated to have critical vulnerabilities [215], [216].

Originally, HALucinator was designed for Bare Metal Embedded Systems (BMES). Other efforts such as Firmadyne and Costin work at the General Purpose Embedded System (GPES) space, meaning that there is no real convenient or usable tool for the Special Purpose Embedded system(SPES) emulation area. To fill this gap, and advance the state of the art, HALucinator and GHALdra were extended to work with SPES, specifically in this case VxWorks. It is important to note that adding support for other SPES operating systems would follow the same procedure.

By adapting the techniques used for HALucinator in intercepting and replacing functionality, we created a Re-hosting Support Layer which is a layer of intercepts at key VxWorks functions to enable file system operations, asynchronous tasks, and interactive communication for ethernet and serial ports. By using this VxWorks RSL, it is possible to partially emulate a Schneider Electric SCADAPack 350 Remote Terminal Unit, a Schneider Electric

Modicon 340 Programmable Logic Controller and a Hughes 9201 BGAN inmarsat terminal. By using these real devices it shows that this approach is tractable for real RE and vulnerability research.

9.1 Background

Firmware that include VxWorks as the operating system have both VxWorks code and device specific code. In this work, the VxWorks versions that were targeted were 5.5 and 6.4. While these may seem old or out-dated (pre 2014), think about the last time that control firmware or hardware was updated at places near you. The stoplights you may drive past, the electric grid components around you, water systems, etc. All of these systems have control units and embedded devices integrated into their systems. You can probably guess at how many of these devices have been replaced since 2014 when VxWorks 7 came out, meaning that there are plenty of devices running VxWorks 5.5 and 6.4. I should also note that the techniques here will work for VxWorks 7, they have just not been tested on it.

Emulating these devices allows for scalable dynamic testing, allowing for vulnerability discovery and research and if the emulation is high enough, we can use the emulation for virtual testbeds. We now discuss how to determine emulation uses and where a given emulation fidelity is on the emulation utility scale.

9.1.1 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 [217]. Our scale goes from zero to five, with zero having the least utility and five having the most. This scale is related to Figure 3.1.

As can be seen in the Figure 9.1, **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

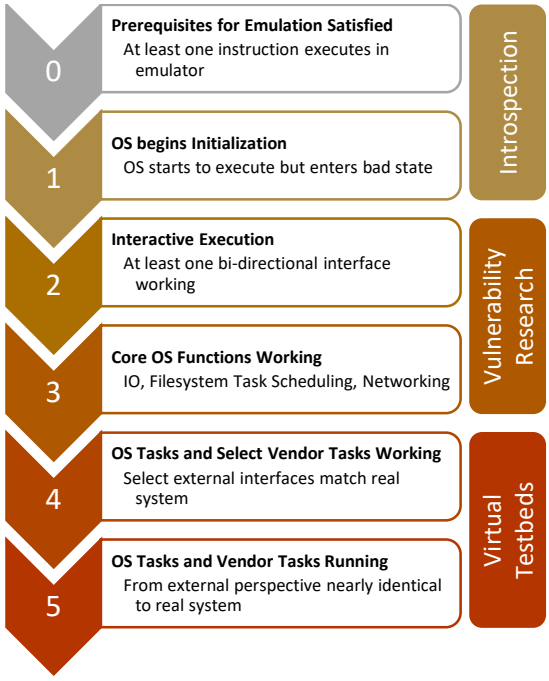


Figure 9.1. Re-hosting Utility Scale

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.

9.1.2 VxWorks

Starting at the top of Figure 9.2 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.

9.2 VxWorks Re-Hosting Support Layer

VxWorks further makes RE simpler by embedding partial symbol tables into the firmware. To retrieve symbols inside of Ghidra we can run the VxWorksSymTab_Finder.java script [218] and it will search the entire binary and parse the symbol tables. VxWorks Manuals [24], [219], [220], programmers guides [221] and headers [222] along with the ARM manual [223] were very helpful to parsing how VxWorks actually behaved in conjunction with ARM, making it possible to RE and re-host firmware. With the knowledge contained in these resources, we somewhat understand the VxWorks design at a sufficient level where I can briefly describe the VxWorks Re-Hosting Support Layer. First logging and error messaging is enabled by capturing *errnoSet* and mapping the error number with a reading string found online [224]. Initialization code that occurs is device specific, and usually occurs from *usrRoot*. Intercepting and replacing hardware interactions at this point is critical to enable re-hosting. When looking at the different hardware interactions available, it is usually

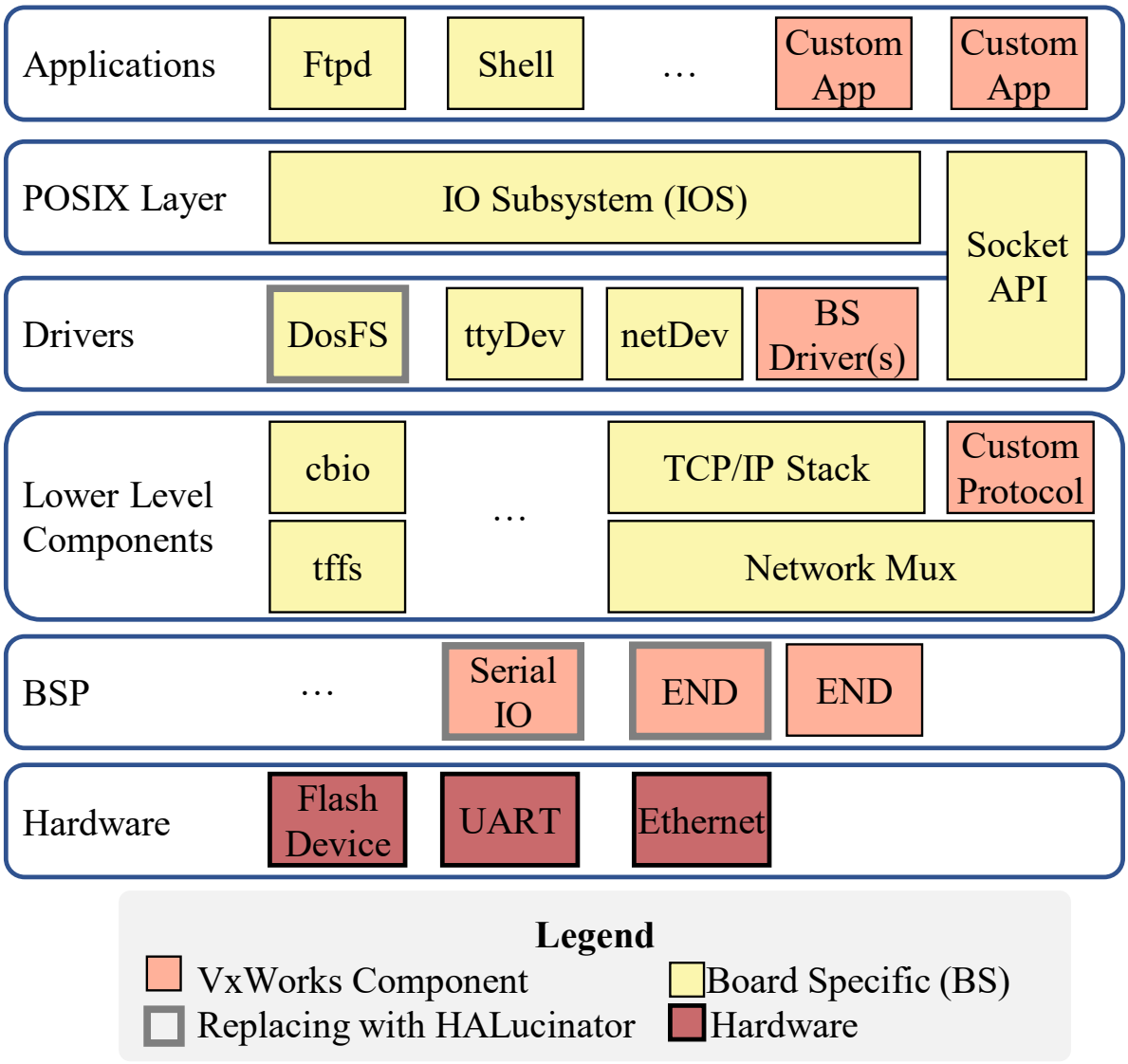


Figure 9.2. VxWorks Layer Diagram. This is an example and is generic. Some of the boxes shown here may change, disappear and others may appear depending on the specific device.

recommended to disable everything, and enable one part at a time to get that component working, delaying any scheduler or scheduling of tasks for as long as possible.

This is when supporting asynchronous tasks comes into play. For GHALdra to work in conjunction with HALucinator I had to allow sending interrupts and triggering the clock. This re-used static analysis work that examined calls to *intConnect* and the call tree of *sysClkInit*.

To reuse other handlers and support from the HALucinator effort in the VxWorks RSL, the changes were very minor, requiring only changing the messaging from the Python implementation of ZMQ to JeroMQ and any Python3 only code to be compatible with Python2.7. Libraries that do not work will with Jython were also removed, including IPython.

10. SUMMARY

Embedded systems will continue to grow in number and applicability, with billions of devices being prevalent in our every day lives. These devices, while helpful and useful bring with them an inherent risk and vulnerability that should be understood. To determine these risks and vulnerabilities I have presented both static and dynamic analysis techniques, and have shown that emulation can truly be used on a large scale in the near future for virtually any device architecture.

10.1 Current State of Things

In Figure 10.1 I replicate the figure from Figure 3.1, but make changes to the current state of things after the inclusion of my work. The figure had to be changed somewhat, as now there are multiple automation levels at the HALucinator and GHALdra bubbles of classification at the conclusion of this work. As can be seen, HALucinator is now more automated, and is classified at the Register Data Fidelity/Basic Block Execution Fidelity and GHALdra is created at the Register Data Fidelity/Instruction Execution Fidelity with a relatively high level of automation. The bubbles for the fidelity of each tool is located at the same position, but now the automation level needs distinguished at the same point on the graph.

While these tools are not all encompassing of related existing tools, I have reviewed the most relevant tools related this work, described their functionalities, and given the practitioner knowledge and flow charts to aid in selecting the right tool for their purpose.

10.2 Future Work

I plan to continue work integrating the PMatch library and function matching into the GHALdra emulation to allow for a dynamic library matching that may work cross platform or for various compilers, versions, and compilation arguments. Continuation of work GHALdra and HALucinator will be useful in my future work at Sandia National Laboratories. I also believe that the area would benefit tremendously from a stand alone Pcode emulator that

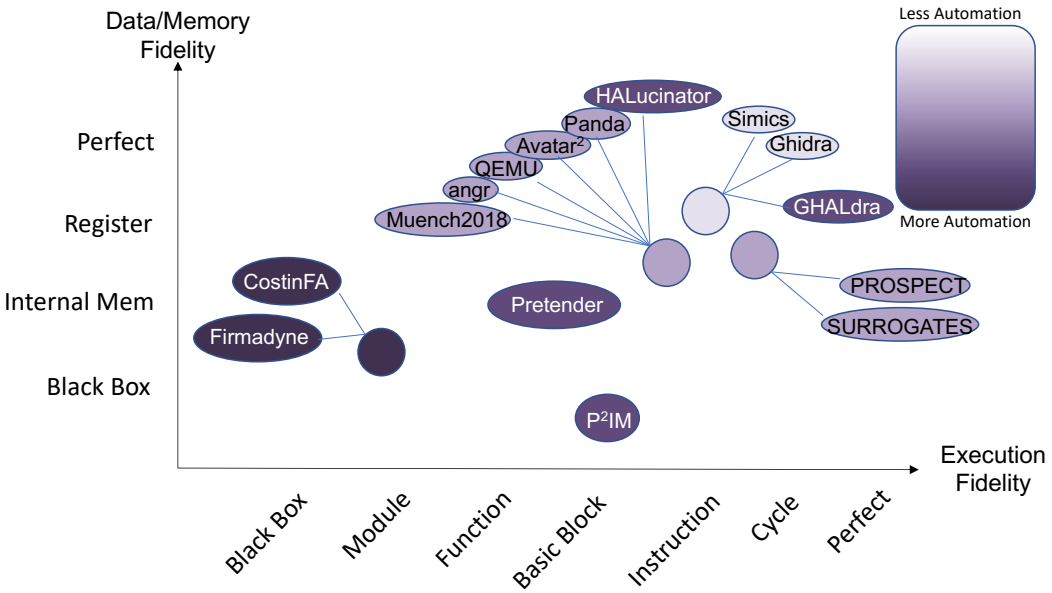


Figure 10.1. Final Classifications for tools after contributions.

does caching and is fast like QEMU. In this way we can have a very fast emulator for virtually all architectures with minimal effort.

10.3 Conclusion

In summary, I have provided a comprehensive review and systematization for the area of system emulation and firmware re-hosting. In Chapter 1, I introduced the motivation for my research and work in emulation, firmware re-hosting and analysis. In Chapter 2, I presented the history and evolution of emulation along with the main base emulators in the space. In Chapter 3, I present classification techniques for analyzing the various tools in the field before classifying the most relevant tools related to my research. In Chapter 4, I do a comprehensive explanation and breakdown of the process of emulating a system and re-hosting a firmware. I have shown how using my contributions of GHALdra, PMatch, HQ-Tracer presented in Chapter 5, Chapter 6, and Chapter 7 repectively, can reduce the amount of effort needed to re-host a firmware. The approach of doing partial emulation is a powerful tool, and is perhaps the most practical real world use of my contributions. In Chapter 8, I explain other scripts that are useful in overcoming emulation challenges, and in Chapter 9, I share how GHALdra integrates the VxWorks Re-hosting Support Layer to enable emulation for Special Purpose Embedded Systems.

I have provided examples showing how to use the software included in this research and show that by using said software the practitioner can successfully emulate and re-host firmware from real world devices. Devices emulated including an STMicroelectonics STM32469I-EVAL firmware, a Schneider Electric SCADAPack 350 remote terminal unit, a Schneider Electric Modicon 340 programmable logic controller, and a Hughes 9201 BGAN inmarsat terminal.

REFERENCES

[1] 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, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/clements.

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

[3] K. Wiles, *First all-digital nuclear reactor system in the u.s. installed at purdue university*, Jul. 2019. [Online]. Available: https://www.purdue.edu/newsroom/releases/2019/Q3/first-all-digital-nuclear-reactor-control-system-in-the-u.s.-installed-at-purdue-university.html.

[4] R. Li, Z. Zhao, X. Zhou, G. Ding, Y. Chen, Z. Wang, and H. Zhang, "Intelligent 5g: When cellular networks meet artificial intelligence," *IEEE Wireless Communications*, vol. 24, no. 5, pp. 175–183, 2017.

[5] U. D. of Energy, *The smart grid*, May 2020. [Online]. Available: https://www.smartgrid.gov/the_smart_grid/smart_grid.html.

[6] P. Gandhi, S. Khanna, and S. Ramaswamy, *Which industries are the most digital (and why)?* Oct. 2017. [Online]. Available: https://hbr.org/2016/04/a-chart-that-shows-which-industries-are-the-most-digital-and-why.

[7] J. Manyika, S. Ramaswamy, S. Khanna, H. Sarrazin, G. Pinkus, G. Sethupathy, and A. Yaffe, *Digital america: A tale of the haves and have-mores*, Dec. 2015. [Online]. Available: https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/digital-america-a-tale-of-the-haves-and-have-mores.

[8] C. Cimpanu, *Android exploits are now worth more than ios exploits for the first time*, Sep. 2019. [Online]. Available: https://www.zdnet.com/article/android-exploits-are-now-worth-more-than-ios-exploits-for-the-first-time/.

[9] C. Reichert, *Google's android bug bounty program will now pay out $1.5 million*, Nov. 2019. [Online]. Available: https://www.cnet.com/news/googles-android-bug-bounty-program-will-now-pay-out-1-5-million/.

[10] B. Hesse, *Earn up to $1 million from apple's expanded bug bounty program*, Aug. 2019. [Online]. Available: https://lifehacker.com/earn-up-to-1-million-from-apples-expanded-bug-bounty-p-1837106598.

[11] L. H. Newman, *Facebook bug bounty program makes biggest reward payout yet*, Dec. 2018. [Online]. Available: https://www.wired.com/story/facebook-bug-bounty-biggest-payout/.

[12] Cisomag, *Tesla offers us$1 million and a car to hack its model 3 car*, Jan. 2020. [Online]. Available: https://www.cisomag.com/tesla-offers-us1-million-and-a-car-as-bug-bounty-reward/.

[13] M. Mickos, *$20m in bounties paid and $100m in sight*, Aug. 2017. [Online]. Available: https://www.hackerone.com/blog/20M-in-bounties-paid-and-100M-in-sight.

[14] S. Karnouskos, "Stuxnet worm impact on industrial cyber-physical system security," in *37th Annual Conference of the IEEE Industrial Electronics Society*, Nov. 2011, pp. 4490–4494. DOI: 10.1109/IECON.2011.6120048.

[15] I. Ahmed, S. Obermeier, M. Naedele, and G. G. Richard III, "Scada systems: Challenges for forensic investigators," *Computer*, vol. 45, no. 12, pp. 44–51, Dec. 2012, ISSN: 1558-0814. DOI: 10.1109/MC.2012.325.

[16] S. Kalle, N. Ameen, H. Yoo, and I. Ahmed, "Clik on plcs! attacking control logic with decompilation and virtual plc," in *Workshop on Binary Analysis Research 2019*, San Diego, California, USA, Feb. 2019. DOI: 10.14722/BAR.2019.23074.

[17] U. Lindqvist and P. G. Neumann, "The future of the internet of things," *Commun. ACM*, vol. 60, no. 2, pp. 26–30, Jan. 2017, ISSN: 0001-0782. DOI: 10.1145/3029589. [Online]. Available: http://doi.acm.org/10.1145/3029589.

[18] K. S. L. Tencent, *Car hacking research: Remote attack tesla motors*, Sep. 2016. [Online]. Available: https://keenlab.tencent.com/en/2016/09/19/Keen-Security-Lab-of-Tencent-Car-Hacking-Research-Remote-Attack-to-Tesla-Cars/.

[19] C. Miller and C. Valasek, *Remote Exploitation of an Unaltered Passenger Vehicle*, Aug. 2015. [Online]. Available: %7Bhttp://illmatics.com/Remote%20Car%20Hacking.pdf%7D.

[20] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *2015 13th Annual Conference on Privacy, Security and Trust*, Jul. 2015, pp. 145–152. DOI: 10.1109/PST.2015.7232966.

[21] 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, ISBN: 978-1-931971-15-7. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin.

[22] M. Yu, J. Zhuge, M. Cao, Z. Shi, and L. Jiang, "A survey of security vulnerability analysis, discovery, detection, and mitigation on iot devices," *Future Internet*, vol. 12, no. 2, p. 27, Feb. 2020, ISSN: 1999-5903. DOI: 10.3390/fi12020027. [Online]. Available: http://dx.doi.org/10.3390/fi12020027.

[23] E. Feng and A. Cheng, *China's tech giant huawei spans much of the globe despite u.s. efforts to ban it*, Oct. 2019. [Online]. Available: https://www.npr.org/2019/10/24/759902041/chinas-tech-giant-huawei-spans-much-of-the-globe-despite-u-s-efforts-to-ban-it.

[24] VxWorks, *Vxworks reference manual : Libraries*, Dec. 2020. [Online]. Available: https://www.ee.ryerson.ca/~courses/ee8205/Data-Sheets/Tornado-VxWorks/vxworks/ref/libIndex.html.

[25] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with panda," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, Los Angeles, CA, USA: ACM, 2015, 4:1–4:11, ISBN: 978-1-4503-3642-0. DOI: 10.1145/2843859.2843867. [Online]. Available: http://doi.acm.org/10.1145/2843859.2843867.

[26] A. Clements, L. Carpenter, W. A. Moeglein, and C. Wright, "Is your firmware real or re-hosted? a case study in re-hosting vxworks control system firmware," *Workshop on Binary Analysis Research (BAR)*, Feb. 2021.

[27] 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.

[28] 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*, Xi'an, China: ACM, 2016, pp. 437–448, ISBN: 978-1-4503-4233-9. DOI: 10.1145/2897845.2897900. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897900.

[29] B. Van Leeuwen, V. Urias, J. Eldridge, C. Villamarin, and R. Olsberg, "Cyber security analysis testbed: Combining real, emulation, and simulation," in *44th Annual 2010 IEEE International Carnahan Conference on Security Technology*, Oct. 2010, pp. 121–126. DOI: 10.1109/CCST.2010.5678720.

[30] C. Kruegel, "Full system emulation: Achieving successful automated dynamic analysis of evasive malware," in *black hat USA 2014 Workshop*, Las Vegas, NV, USA: blackhat.com, Aug. 2014. [Online]. Available: https://www.blackhat.com/docs/us-14/materials/us-14-Kruegel-Full-System-Emulation-Achieving-Successful-Automated-Dynamic-Analysis-Of-Evasive-Malware-WP.pdf.

[31] F. D. Tanasache, M. Sorella, S. Bonomi, R. Rapone, and D. Meacci, "Building an emulation environment for cyber security analyses of complex networked systems," *Proceedings of the 20th International Conference on Distributed Computing and Networking*, 2019. DOI: 10.1145/3288599.3288618.

[32] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2020.

[33] S. Shah, *The arm-x firmware emulation framework*, May 2020. [Online]. Available: https://github.com/therealsaumil/armx.

[34] J. Viega and H. Thompson, "The state of embedded-device security (spoiler alert: It's bad)," *IEEE Symposium on Security and Privacy*, vol. 10, no. 5, pp. 68–70, Sep. 2012. DOI: 10.1109/MSP.2012.134.

[35] J. Hall, *Hp laserjet the early history*. [Online]. Available: http://hparchive.com/seminar%5C_notes/HP%5C_LaserJet%5C_The%5C_Early%5C_History%5C_by%5C_Jim%5C_Hall%5C_110512.pdf.

[36] A. Kaluszka, *Computer emulation history*. [Online]. Available: https://kaluszka.com/vt/emulation/history.html.

[37] R. Phillips and B. Montalvo, "Using emulation to debug control logic code," *Proceedings of the 2010 Winter Simulation Conference*, 2010. DOI: 10.1109/wsc.2010.5678904.

[38] K. P. Lawton, "Bochs: A portable pc emulator for unix/x," *Linux J.*, vol. 1996, no. 29es, Sep. 1996, ISSN: 1075-3583. [Online]. Available: http://dl.acm.org/citation.cfm?id=326350.326357.

[39] DOSBox, *DOSBox*, Sep. 2019. [Online]. Available: https://www.dosbox.com/.

[40] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Bharadwaj Yadavalli, and J. Yates, "Fx!32 a profile-directed binary translator," *IEEE Micro*, vol. 18, no. 2, pp. 56–64, Mar. 1998. DOI: 10.1109/40.671403.

[41] PCem, *PCem*, Sep. 2019. [Online]. Available: https://github.com/Anamon/pcem.

[42] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[43] M. F. Thompson and T. Vidas, *Cgc monitor: A vetting system for the darpa cyber grand challenge*, 2018. [Online]. Available: https://calhoun.nps.edu/handle/10945/59209.

[44] NSA, *Ghidra*, version 9.0.4, Sep. 2019. [Online]. Available: https://ghidra-sre.org/.

[45] Sickendick, Karl, *pcode-emulator*, Nov. 2019. [Online]. Available: https://github.com/kc0bfv/pcode-emulator.

[46] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252. [Online]. Available: http://doi.acm.org/10.1145/360248.360252.

[47] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 1066–1071, ISBN: 9781450304450. DOI: 10.1145/1985793.1985995. [Online]. Available: https://doi.org/10.1145/1985793.1985995.

[48] BE-PUM, *BE-PUM*, Sep. 2019. [Online]. Available: https://github.com/NMHai/BE-PUM.

[49] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion, "Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2016, pp. 653–656. DOI: 10.1109/SANER.2016.43. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER.2016.43.

[50] B. Loring, D. Mitchell, and J. Kinder, "Expose: Practical symbolic execution of standalone javascript," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, Santa Barbara, CA, USA: ACM, 2017, pp. 196–199, ISBN: 978-1-4503-5077-8. DOI: 10.1145/3092282.3092295. [Online]. Available: http://doi.acm.org/10.1145/3092282.3092295.

[51] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song, "Transformation-aware exploit generation using a hi-cfg," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-85, May 2013. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-85.html.

[52] BitBlaze, *FuzzBALL*, Sep. 2019. [Online]. Available: https://github.com/bitblaze-fuzzball/fuzzball.

[53] Samsung, *Jalangi2*, Sep. 2019. [Online]. Available: https://github.com/Samsung/jalangi2.

[54] Janala2, *Janala2*, Sep. 2019. [Online]. Available: https://github.com/ksen007/janala2.

[55] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, "Javert 2.0: Compositional symbolic execution for javascript," *Proceedings of the ACM on Principles of Programming Languages*, vol. 3, 66:1–66:31, Jan. 2019, ISSN: 2475-1421. DOI: 10.1145/3290379. [Online]. Available: http://doi.acm.org/10.1145/3290379.

[56] P. Braione, G. Denaro, and M. Pezzè, "Symbolic execution of programs with heap inputs," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy: ACM, 2015, pp. 602–613, ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786842. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786842.

[57] P. Braione, G. Denaro, and M. Pezzè, "Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, Saint Petersburg, Russia: ACM, 2013, pp. 411–421, ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491433. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491433.

[58] Sen, Koushik, *jCUTE*, Sep. 2019. [Online]. Available: https://github.com/osl/jcute.

[59] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, San Diego, California: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[60] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, *Manticore: A user-friendly symbolic execution framework for binaries and smart contracts*, 2019. arXiv: 1907.03890 [cs.SE].

[61] ConsenSys, *Mythril*, Sep. 2019. [Online]. Available: https://github.com/ConsenSys/mythril.

[62] A. Sharma, "Exploiting undefined behaviors for efficient symbolic execution," in *Companion Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India: ACM, 2014, pp. 727–729, ISBN: 978-1-4503-2768-8. DOI: 10.1145/2591062.2594450. [Online]. Available: http://doi.acm.org/10.1145/2591062.2594450.

[63] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 265–278, Mar. 2011, ISSN: 0163-5964. DOI: 10.1145/1961295.1950396. [Online]. Available: http://doi.acm.org/10.1145/1961295.1950396.

[64] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symposium on Security and Privacy*, Washington, DC, USA: IEEE Computer Society, 2012, pp. 380–394, ISBN: 978-0-7695-4681-0. DOI: 10.1109/SP.2012.31. [Online]. Available: https://doi.org/10.1109/SP.2012.31.

[65] 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 *Proceedings of the 2016 Network and Distributed System Security Symposium*, 2016.

[66] AFL-Fuzz, *Afl-fuzz*, Sep. 2019. [Online]. Available: https://github.com/google/AFL.

[67] radamsa, *Radamsa*, Sep. 2019. [Online]. Available: https://gitlab.com/akihe/radamsa.

[68] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium*, 2018, pp. 745–761.

[69] K. Serebryany, *Oss-fuzz-google's continuous fuzzing service for open source software*, 2017.

[70] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing.," in *Network and Distributed Systems Security Symposium*, vol. 17, 2017, pp. 1–14.

[71] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of for kernelspace fuzzing," in *13th USENIX Workshop on Offensive Technologies*, Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/maier.

[72] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. L. Cheong, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *25th Annual Network and Distributed System Security Symposium, 2018, San Diego, California, USA, February 18-21, 2018*, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018%5C_01A-1%5C_Chen%5C_paper.pdf.

[73] Google, *honggfuzz*, Sep. 2019. [Online]. Available: https://github.com/google/honggfuzz.

[74] Google, *clusterfuzz*, Sep. 2019. [Online]. Available: https://github.com/google/clusterfuzz.

[75] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Network and Distributed Systems Security Symposium*, 2008.

[76] Google, *Syzkaller*, Sep. 2019. [Online]. Available: https://github.com/google/syzkaller.

[77] X. Mendez, *Wfuzz*, Sep. 2019. [Online]. Available: https://github.com/xmendez/wfuzz.

[78] Google, *winafl*, Sep. 2019. [Online]. Available: https://github.com/googleprojectzero/winafl.

[79] G. Grieco, M. Ceresa, and P. Buiras, "Quickfuzz: An automatic random fuzzer for common file formats," in *Proceedings of the 9th International Symposium on Haskell*, Nara, Japan: ACM, 2016, pp. 13–20, ISBN: 978-1-4503-4434-0. DOI: 10.1145/2976002.2976017. [Online]. Available: http://doi.acm.org/10.1145/2976002.2976017.

[80] boofuzz, *Boofuzz*, Sep. 2019. [Online]. Available: https://github.com/jtpereyda/boofuzz.

[81] Google, *Domato*, Sep. 2019. [Online]. Available: https://github.com/googleprojectzero/domato.

[82] SSRFmap, *Ssrfmap*, Sep. 2019. [Online]. Available: https://github.com/swisskyrepo/SSRFmap.

[83] Google, *fuzzilli*, Sep. 2019. [Online]. Available: https://github.com/googleprojectzero/fuzzilli.

[84] D. Jones, "Trinity: A system call fuzzer," in *Proceedings of the 13th Ottawa Linux Symposium*, 2011.

[85] Google, *Gofuzz*, Sep. 2019. [Online]. Available: https://github.com/google/gofuzz.

[86] C. Domas, "Breaking the x86 isa," in *black hat USA 2017 Workshop*, Las Vegas, NV, USA: blackhat.com, Jul. 2017. [Online]. Available: https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-Instruction-Set-wp.pdf.

[87] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 2123–2138.

[88] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *IEEE Symposium on Security and Privacy*, IEEE, 2018, pp. 697–710.

[89] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "Firmfuzz: Automated iot firmware introspection and analysis," *Proceedings Of The 2nd International ACM Workshop On Security And Privacy For The Internet-Of-Things*, pp. 15–21, 2019. DOI: 10.1145/3338507.3358616. [Online]. Available: http://infoscience.epfl.ch/record/276976.

[90] 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.

[91] 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, ISBN: 1-891562-35-5. DOI: 10.14722/ndss.2014.23229.

[92] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits," in *IEEE Symposium on Security and Privacy*, 2017.

[93] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments," in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[94] R. Wang, Y. Shoshitaishvili, A. Bianchi, M. Aravind, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making Reassembly Great Again," in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[95] Shellphish, *Cyber grand shellphish*, 2017. [Online]. Available: http://phrack.org/papers/cyber%5C_grand%5C_shellphish.html.

[96] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proceedings of the 2015 Network and Distributed System Security Symposium*, 2015.

[97] R. Parvez, P. A. S. Ward, and V. Ganesh, "Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries," in *26th Annual International Conference on Computer Science and Software Engineering*, Toronto, Ontario, Canada: IBM Corp., 2016, pp. 116–127. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049877.3049889.

[98] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 709–724.

[99] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz, "Dynamic Hooks: Hiding Control Flow Changes Within Non-Control Data," in *23rd USENIX Security Symposium*, 2014, pp. 813–328.

[100] M. Tancreti, M. S. Hossain, S. Bagchi, and V. Raghunathan, "Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems," in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, ACM, 2011, pp. 288–301.

[101] V. Sundaram, P. Eugster, and X. Zhang, "Efficient diagnostic tracing for wireless sensor networks," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, ACM, 2010, pp. 169–182.

[102] M. Tancreti, V. Sundaram, S. Bagchi, and P. Eugster, "Tardis: Software-only system-level record and replay in wireless sensor networks," in *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, ACM, Seattle, Washington: ACM, 2015, pp. 286–297, ISBN: 978-1-4503-3475-4. DOI: 10.1145/2737095.2737096. [Online]. Available: http://doi.acm.org/10.1145/2737095.2737096.

[103] 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*, San Diego (USA), Feb. 2018.

[104] 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.

[105] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: Peripheral proxying supported embedded code testing," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, Kyoto, Japan: ACM, 2014, pp. 329–340, ISBN: 978-1-4503-2800-5. DOI: 10.1145/2590296.2590301. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590301.

[106] K. Koscher, T. Kohno, and D. Molnar, "Surrogates: Enabling near-real-time dynamic analyses of embedded systems," in *9th USENIX Workshop on Offensive Technologies*, Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/koscher.

[107] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling (extended version)," *ArXiv*, vol. abs/1909.06 2019.

[108] 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.

[109] Vector 35, *Binary Ninja*, Sep. 2019. [Online]. Available: https://binary.ninja/.

[110] J. Broome and D. Marx, *Method and iimplementation for intercepting and processing system calls in programmed digital computer to emulate retrograde operating system*, US Patent 6,086,623, Jul. 2000.

[111] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, Springer, 2008, pp. 1–25.

[112] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*, Springer, 2011, pp. 463–469.

[113] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in *20th Annual Computer Security Applications Conference*, IEEE, 2004, pp. 91–100.

[114] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *14th USENIX Security Symposium*, vol. 14, 2005, pp. 11–11.

[115] R. Team, *Radare2 Book*. GitHub, 2017.

[116] Hemel, Armijn, *BANG - Binary Analysis Next Generation*, Sep. 2019. [Online]. Available: https://github.com/armijnhemel/binaryanalysis-ng.

[117] ReFirm Labs, *binwalk*, Sep. 2019. [Online]. Available: https://github.com/ReFirmLabs/binwalk.

[118] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, London, United Kingdom: ACM, 2007, pp. 196–206, ISBN: 978-1-59593-734-6. DOI: 10.1145/1273463.1273490. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273490.

[119] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *22nd USENIX Security Symposium*, Washington, D.C.: USENIX Association, Aug. 2013, pp. 463–478, ISBN: 978-1-931971-03-4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson.

[120] firmware-mod-kit, *firmware-mod-kit*, Sep. 2019. [Online]. Available: https://github.com/rampageX/firmware-mod-kit.

[121] G. Hernandez, F. Fowze, D. (Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2245–2262, ISBN: 9781450349468. DOI: 10.1145/3133956.3134050. [Online]. Available: https://doi.org/10.1145/3133956.3134050.

[122] S. Thomas, F. Garcia, and T. Chothia, "Humidify: A tool for hidden functionality detection in firmware," Jun. 2017, pp. 279–300, ISBN: 978-3-319-60875-4. DOI: 10.1007/978-3-319-60876-1_13.

[123] A. Keliris and M. Maniatakos, "Icsref: A framework for automated reverse engineering of industrial control systems binaries," in *Network and Distributed Systems Security Symposium*, Feb. 2019.

[124] H. Rays. [Online]. Available: https://hex-rays.com/products/ida/.

[125] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *27th USENIX Security Symposium*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 309–326, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani.

[126] M. A. B. Khadra, D. Stoffel, and W. Kunz, "Speculative disassembly of binary code," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Pittsburgh, Pennsylvania: ACM, 2016, 16:1–16:10, ISBN: 978-1-4503-4482-1. DOI: 10.1145/2968455.2968505. [Online]. Available: http://doi.acm.org/10.1145/2968455.2968505.

[127] Avast Software, *RetDec: A retargetable machine-code decompiler*, https://retdec.com/.

[128] Y. Li, J. M. McCune, and A. Perrig, "Viper: Verifying the integrity of peripherals' firmware," in *18th ACM Conference on Computer and Communications Security*, Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 3–16, ISBN: 9781450309486. DOI: 10.1145/2046707.2046711. [Online]. Available: https://doi.org/10.1145/2046707.2046711.

[129] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718.

[130] Capstone, *Capstone Disassembler*, Sep. 2019. [Online]. Available: http://www.capstone-engine.org/.

[131] A. Costin and J. Zaddach, "Embedded devices security and firmware reverse engineering," in *black hat USA 2013 Workshop*, Las Vegas, NV, USA: blackhat.com, Jul. 2013. [Online]. Available: https://media.blackhat.com/us-13/US-13-Zaddach-Workshop-on-Embedded-Devices-Security-and-Firmware-Reverse-Engineering-WP.pdf.

[132] OWASP, *Iotgoat*, May 2020. [Online]. Available: https://github.com/OWASP/IoTGoat.

[133] Wireshark, *Wireshark*, Sep. 2019. [Online]. Available: https://www.wireshark.org/.

[134] EtherApe, *EtherApe*, Sep. 2019. [Online]. Available: https://etherape.sourceforge.io/.

[135] TCPDump, *TCPDump*, Sep. 2019. [Online]. Available: http://www.tcpdump.org/.

[136] Netresec, *NetworkMiner*, Sep. 2019. [Online]. Available: https://www.netresec.com/?page=NetworkMiner.

[137] Kismet, *Kismet*, Sep. 2019. [Online]. Available: https://www.kismetwireless.net/.

[138] Telerik, *Fiddler*, Sep. 2019. [Online]. Available: https://www.telerik.com/fiddler.

[139] NetWorkPacketCapture, *NetWorkPacketCapture*, Sep. 2019. [Online]. Available: https://github.com/huolizhuminh/NetWorkPacketCapture.

[140] Cisco, *Joy*, Sep. 2019. [Online]. Available: https://github.com/cisco/joy.

[141] PixelCyber, *Thor*, Sep. 2019. [Online]. Available: https://github.com/PixelCyber/Thor.

[142] PcapPlusPlus, *PcapPlusPlus*, Sep. 2019. [Online]. Available: https://github.com/seladb/PcapPlusPlus.

[143] Malcolm, *Malcolm*, Sep. 2019. [Online]. Available: https://github.com/idaholab/Malcolm.

[144] DroidSniff, *DroidSniff*, Sep. 2019. [Online]. Available: https://github.com/evozi/DroidSniff.

[145] 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, ISBN: 978-3-030-15462-2.

[146] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas, "Implementation and implications of a stealth hard-drive backdoor," in *Proceedings of the 29th Annual Computer Security Applications Conference*, New Orleans, Louisiana, USA: Association for Computing Machinery, 2013, pp. 279–288, ISBN: 9781450320153. DOI: 10.1145/2523649.2523661. [Online]. Available: https://doi.org/10.1145/2523649.2523661.

[147] FaceDancer, *FaceDancer*, Apr. 2019. [Online]. Available: https://github.com/usb-tools/Facedancer.

[148] 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, S68–S76, 2015, The Proceedings of the Fifteenth Annual DFRWS Conference, ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2015.05.008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287615000535.

[149] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, *Firmware.re*, May 2020. [Online]. Available: http://firmware.re/usenixsec14/.

[150] firminsight, *Firminsight*, May 2020. [Online]. Available: https://github.com/ilovepp/firminsight.

[151] T. Vea, *Firmwaredb*, May 2020. [Online]. Available: https://github.com/kvisle/firmwaredb.

[152] C. Schultz, *Firmware_collection*, May 2020. [Online]. Available: https://github.com/f47h3r/firmware%5C_collection.

[153] Praetorian, *The damn vulnerable router firmware project*, May 2020. [Online]. Available: https://github.com/praetorian-code/DVRF.

[154] angr, *boyscout*, Sep. 2019. [Online]. Available: https://github.com/angr/angr/blob/master/angr/analyses/boyscout.py.

[155] J. Clemens, "Automatic classification of object code using machine learning," *Digital Investigation*, vol. 14, no. S1, S156–S162, Aug. 2015, ISSN: 1742-2876. DOI: 10.1016/j.diin.2015.05.007. [Online]. Available: https://doi.org/10.1016/j.diin.2015.05.007.

[156] P. De Nicolao, M. Pogliani, M. Polino, M. Carminati, D. Quarta, and S. Zanero, "Elisa: Eliciting isa of raw binaries for fine-grained code and data separation," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Giuffrida, S. Bardin, and G. Blanc, Eds., Cham: Springer International Publishing, 2018, pp. 351–371, ISBN: 978-3-319-93411-2.

[157] S. Kairajärvi, A. Costin, and T. Hämäläinen, "Isadetect: Usable automated detection of cpu architecture and endianness for executable binary files and object code," in *Tenth ACM Conference on Data and Application Security and Privacy*, New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 376–380, ISBN: 9781450371070. DOI: 10.1145/3374664.3375742. [Online]. Available: https://doi.org/10.1145/3374664.3375742.

[158] H. Xue, S. Sun, G. Venkataramani, and T. Lan, "Machine learning-based analysis of program binaries: A comprehensive study," *IEEE Access*, vol. Volume 7, pp. 65 889–65 912, 2019.

[159] 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, ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2016.01.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287616000037.

[160] R. Zhu, B. Zhang, J. Mao, Q. Zhang, and Y.-a. Tan, "A methodology for determining the image base of arm-based industrial control system firmware," *International Journal of Critical Infrastructure Protection*, vol. Volume 16, pp. 26–35, 2017, ISSN: 1874-5482. DOI: https://doi.org/10.1016/j.ijcip.2016.12.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1874548216300014.

[161] angr, *girlscout*, Sep. 2019. [Online]. Available: https://github.com/angr/angr/blob/master/angr/analyses/girlscout.py.

[162] PAGalaxyLab, *vxhunter*, Oct. 2019. [Online]. Available: https://github.com/PAGalaxyLab/vxhunter.

[163] Leveldown Security, *SVD-Loader-Ghidra*, Dec. 2019. [Online]. Available: https://github.com/leveldown-security/SVD-Loader-Ghidra.

[164] T. Reed, *Subzero*, May 2020. [Online]. Available: https://github.com/theopolis/subzero.

[165] A. Hemel and S. Coughlan, *Binary analysis toolkit*, May 2020. [Online]. Available: http://www.binaryanalysis.org/old/home.

[166] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: Detecting the taint-style vulnerability in embedded device firmware," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2018, pp. 430–441. DOI: 10.1109/DSN.2018.00052.

[167] X. Wang, R. Ma, B. Dou, Z. Jian, and H. Chen, "OFFDTAN: A New Approach of Offline Dynamic Taint Analysis for Binaries," *Security and Communication Networks*, vol. Volume 2018, p. 13, 2018. [Online]. Available: 10.1155/2018/7693861.

[168] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti, "Pie: Parser identification in embedded systems," in *Proceedings of the 31st Annual Computer Security Applications Conference*, Los Angeles, CA, USA: Association for Computing Machinery, 2015, pp. 251–260, ISBN: 9781450336826. DOI: 10.1145/2818000.2818035. [Online]. Available: https://doi.org/10.1145/2818000.2818035.

[169] M. Vasut, *Adding new architecture to qemu*, Jun. 2017. [Online]. Available: https://events17.linuxfoundation.org/sites/events/files/slides/ossj-2017.pdf.

[170] NationalSecurityAgency, *Nationalsecurityagency/ghidra*. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra/wiki/Frequently-asked-questions.

[171] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: Cryptographic function identification in obfuscated binary programs," in *ACM Conference on Computer and Communications Security*, ACM, 2012, pp. 169–182.

[172] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *23rd USENIX Security Symposium*, 2014, pp. 845–860.

[173] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium*, 2015, pp. 611–626.

[174] E. R. Jacobson, N. Rosenblum, and B. P. Miller, "Labeling library functions in stripped binaries," in *10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, ACM, 2011, pp. 1–8.

[175] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *IEEE Symposium on Security and Privacy*, IEEE, 2017, pp. 921–937.

[176] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *13th USENIX Security Symposium*, vol. 13, 2004, pp. 18–18.

[177] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium*, 2017, pp. 99–116.

[178] 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.

[179] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Fossil: A resilient and efficient system for identifying foss functions in malware binaries," *ACM Transactions on Privacy and Security*, vol. 21, no. 2, p. 8, 2018.

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

[181] Y. Liao, R. Cai, G. Zhu, Y. Yin, and K. Li, "Mobilefindr: Function similarity identification for reversing mobile binaries," in *European Symposium on Research in Computer Security*, Springer, 2018, pp. 66–83.

[182] R. Qiao and R. Sekar, "Effective function recovery for cots binaries using interface verification," Technical report, Secure Systems Lab, Stony Brook University, Tech. Rep., 2016.

[183] G. Mittal, D. Zaretsky, G. Memik, and P. Banerjee, "Automatic extraction of function bodies from software binaries," in *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, IEEE, vol. 2, 2005, pp. 928–931.

[184] Sibyl, *Sibyl*, Sep. 2019. [Online]. Available: https://github.com/cea-sec/Sibyl.

[185] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization*, 2004, pp. 75–86.

[186] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100, ISBN: 9781595936332. DOI: 10.1145/1250734.1250746. [Online]. Available: https://doi.org/10.1145/1250734.1250746.

[187] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis," Zynamics, 2009. [Online]. Available: https://static.googleusercontent.com/media/www.zynamics.com/en//downloads/csw09.pdf.

[188] Craig, *Emulating nvram in qemu*, Mar. 2012. [Online]. Available: http://www.devttys0.com/2012/03/emulating-nvram-in-qemu/.

[189] Firmadyne, *Firmadyne/libnvram*, Aug. 2018. [Online]. Available: https://github.com/firmadyne/libnvram.

[190] A. Costin, A. Zarras, and A. Francillon, "Towards automated classification of firmware images and identification of embedded devices," in *ICT Systems Security and Privacy Protection*, S. De Capitani di Vimercati and F. Martinelli, Eds., Cham: Springer International Publishing, 2017, pp. 233–247, ISBN: 978-3-319-58469-0.

[191] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Toronto, Canada: ACM, 2018, pp. 2123–2138, ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243804. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243804.

[192] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE Symposium on Security and Privacy*, Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331, ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.26. [Online]. Available: https://doi.org/10.1109/SP.2010.26.

[193] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, 50:1–50:39, May 2018, ISSN: 0360-0300. DOI: 10.1145/3182657. [Online]. Available: http://doi.acm.org/10.1145/3182657.

[194] P. Liu, C. Xiang, X. Wang, B. Xia, Y. Liu, W. Wang, and Q. Yao, "A noc emulation/verification framework," in *Sixth International Conference on Information Technology: New Generations*, IEEE, 2009, pp. 859–864.

[195] Slack, *Slack*, Sep. 2019. [Online]. Available: https://angr.slack.com.

[196] Comsecuris, *GDB Ghidra*, Sep. 2019. [Online]. Available: https://github.com/Comsecuris/gdbghidra.

[197] H. Rays, *Ida f.l.i.r.t. technology: In-depth*. [Online]. Available: https://www.hex-rays.com/products/ida/tech/flirt/in_depth/.

[198] E. Gustafson, *Libmatch*, Dec. 2020. [Online]. Available: https://github.com/subwire/libmatch.

[199] J. Koret, *Diaphora: A free and open source program diffing tool*, 2015. [Online]. Available: https://github.com/joxeankoret/diaphora.

[200] C. Karamitas and A. Kehagias, "Efficient features for function matching between binary executables," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 335–345. DOI: 10.1109/SANER.2018.8330221.

[201] C. Percival, "Binary diff/patch utility," *URL: http://www. daemonology. net/bsdiff*, 2003.

[202] J. Heirbaut, *Jojodiff - diff utility for binary files*, Jun. 2002. [Online]. Available: http://jojodiff.sourceforge.net/.

[203] sisong, *HDiffPatch*, Sep. 2019. [Online]. Available: https://github.com/sisong/HDiffPatch.

[204] J. P. MacDonald, "Xdelta: Open-source binary diff, differential compression tools, vcdiff (rfc 3284) delta compression," *http://xdelta. org/*,

[205] syscall7, *Machine Emulation With Ghidra*, Mar. 2021. [Online]. Available: https://syscall7.com/machine-emulation-with-ghidra/.

[206] TheRomanXpl0it, *ghidra-emu-fun*, Mar. 2021. [Online]. Available: https://github.com/TheRomanXpl0it/ghidra-emu-fun.

[207] ZeroMQ, *JeroMQ*, Mar. 2021. [Online]. Available: https://github.com/zeromq/jeromq.

[208] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys*, 2021.

[209] Wind River, *Functional safety*, Dec. 2020. [Online]. Available: https://www.windriver.com/functionalsafety.

[210] J. F. Ready, "Vrtx: A real-time operating system for embedded microprocessor applications," *IEEE Micro*, vol. 6, no. 4, pp. 8–17, 1986. DOI: 10.1109/MM.1986.304774.

[211] *Command & data-handling systems*, Dec. 2020. [Online]. Available: https://mars.nasa.gov/mro/mission/spacecraft/parts/command/.

[212] 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.

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

[214] A. Volosincu, *Vxworks: Past and future*, Jul. 2018. [Online]. Available: https://blogs.windriver.com/wind%5C_river%5C_blog/2018/07/vxworks-past-and-future/.

[215] *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.

[216] Wind River, *Security vulnerability response information: Tcp/ip network stack (ipnet, urgent/11)*, Dec. 2020. [Online]. Available: https://www.windriver.com/security/announcements/tcp-ip-network-stack-ipnet-urgent11/.

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

[218] NSA, *Vxworkssymtab_finder.java*, Dec. 2020. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra/blob/fe8d863c47f79d904c10c2c49d16ea4c1b674020/Ghidra/Features/GnuDemangler/%20ghidra%5C_scripts/VxWorksSymTab%5C_Finder.java.

[219] *Vxworks network protocol toolkit user's guide*, Wind River, 500 Wind River Way, Alameda, CA, 94501, Aug. 2002.

[220] *Vxworks network programmer's guide 5.5*, Wind River, 500 Wind River Way, Alameda, CA, 94501, Aug. 2002. [Online]. Available: http://www.ing.iac.es/~docs/external/vxworks.old/Network-Guide-5.5.pdf.

[221] *Vxworks kernel programmer's guide 6.2*, Wind River, 500 Wind River Way, Alameda, CA, 94501, Oct. 2005.

[222] E. Perez, *400plus/iolib.h*, Dec. 2020. [Online]. Available: https://github.com/400plus/400plus/blob/master/vxworks/ioLib.h.

[223] A. Limited, *Arm architecture reference manual*, English, version Version 5t, ARM, Jul. 2005, 1138 pp. [Online]. Available: https://developer.arm.com/documentation/ddi0100/latest/, July, 2005.

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

VITA

Christopher Wright is a Ph.D. candidate in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, Indiana, USA and is advised by Professor Milind Kulkarni. He obtained his Bachelor's of Science (BS) in Computer Engineering from Utah Valley University in 2015. He is a member of the PLCL laboratory at Purdue, is a member of ACM, IEEE, and Eta Kappa Nu (HKN). During his tenure as a student at Purdue he has collaborated with and worked at Lawrence Livermore National Laboratories, Pacific Northwest National Laboratories, Sandia National Laboratories and Facebook. He is interested in the broad areas of systems research and cyber-security, including emulation, embedded systems, distributed computing, compilers, and programming languages.