Jetset: Targeted Firmware Rehosting for Embedded Systems

Evan Johnson, *UC San Diego*
Maxwell Bland, *University of Illinois*
YiFei Zhu, *University of Illinois*
Joshua Mason, *University of Illinois*
Stephen Checkoway, *Oberlin College*
Stefan Savage, *UC San Diego*
Kirill Levchenko, *University of Illinois*

Abstract

The ability to execute code in an emulator is a fundamental part of modern vulnerability testing. Unfortunately, this poses a challenge for many embedded systems, where firmware expects to interact with hardware devices specific to the target. Getting embedded system firmware to run outside its native environment, termed *rehosting*, requires emulating these hardware devices with enough accuracy to convince the firmware that it is executing on the target hardware. However, full fidelity emulation of target devices (which requires considerable engineering effort) may not be necessary to boot the firmware to a point of interest for an analyst (for example, a point where fuzzer input can be injected). We hypothesized that, for the firmware to boot successfully, it is sufficient to emulate only the behavior expected by the firmware, and that this behavior could be inferred automatically.

To test this hypothesis, we developed and implemented Jetset, a system that uses symbolic execution to infer what behavior firmware expects from a target device. Jetset can then generate device models for hardware peripherals in C, allowing an analyst to boot the firmware in an emulator (e.g., QEMU). We successfully applied Jetset to thirteen distinct pieces of firmware together representing three architectures, three application domains (power grid, avionics, and consumer electronics), and five different operating systems. We also demonstrate how Jetset-assisted rehosting facilitates fuzz-testing, a common security analysis technique, on an avionics embedded system, in which we found a previously unknown privilege escalation vulnerability.

1 Introduction

Executing code in a controlled environment is a fundamental part of modern systems analysis. Unfortunately, embedded systems pose a challenge because their code expects to interact with specialized on-chip and off-chip peripherals, such as general-purpose I/O (GPIO) ports, sensors, and communication interfaces. The execution environment must emulate these devices with sufficient fidelity to ensure that observed behavior accurately mimics the target system running on hardware. However, because of the large variety of peripheral devices, most are not modeled by the execution environment, creating a considerable blind spot for our most powerful analysis techniques. Indeed, there may be no documentation at all about a target system, which makes building a complete emulator for it nearly impossible.

In many cases, however, the code of interest to the system analyst is not the code that interacts with peripherals. While peripherals cannot be ignored completely—hardware initialization must appear successful for the system to boot successfully—correct behavior of all devices may not be necessary. For example, an analyst interested in how a target responds to network traffic may not require the execution environment to faithfully model all aspects of the system's GPIO ports or other communication interfaces.

The subject of this work is Jetset, a system that performs *targeted rehosting* of firmware—it automatically infers the expected behavior of embedded system peripherals using only its firmware and then synthesizes a model of the peripherals sufficient to boot to security-critical code of interest. The synthesized peripheral model can then be used in an emulator—in our evaluation we used QEMU [4]—to emulate the hardware environment. An analyst can then use her tool of choice to interact with the firmware. For example, a vulnerability analyst can use Jetset to fuzz-test the system to see how it responds to malformed or otherwise malicious input. More advanced dynamic analyses, like symbolic execution, are also available to the analyst.

Jetset infers the values that need to be read from peripheral devices needed for the program to reach an analyst-specified *goal address*. For example, on the Raspberry Pi target used in our evaluation, our goal is to reach the address where the code jumps to user space. Our key insight is that firmware code interacting with a peripheral device implicitly encodes how the device must behave for the system to boot. Jetset uses symbolic execution of the firmware—specifically the angr framework [31]—to infer data returned from devices. Our technique mitigates path explosion using *guided* symbolic execution in which execution paths are selected using a variation of Tabu Search [19] while minimizing the distance to the goal.

The input to Jetset is the executable firmware image, the firmware entry point (where to start execution), a goal address (the address we want to reach in execution), and a memory layout specifying which parts of the address space represent RAM and which represent memory-mapped I/O. Jetset only requires emulation support for the CPU architecture; it does not require any special hardware, and does not use the underlying hardware device.

To evaluate Jetset, we use it to infer and instantiate peripheral devices for thirteen targets: an aircraft Communication

Management Unit (AMD 486-based system) used on the Boeing 737, Linux on a Raspberry Pi 2 (ARM-based SoC board), the first-stage bootloader on a BeagleBoard-xM (ARM-based SoC board), a SEL-751 Feeder Protection Relay (Motorola ColdFire-based system), and the 9 publicly available real-world targets from prior comparable work[18]. These targets are diverse—they come from 3 different architectures, 5 different operating systems (as well as 3 different bare metal systems), and several different application domains. For each target, Jetset infers the behavior of its peripherals needed for the firmware to complete its boot sequence. Next, Jetset produces C code suitable for use with QEMU that simulates the inferred devices. We then run the firmware in QEMU configured for the target CPU architecture using our synthetic peripherals to complete the configuration.

For two of our targets, we confirm that the synthesized devices work correctly by comparing the emulated system against a reference. For the Raspberry Pi 2, we compare the emulated behavior of the system to its behavior on actual hardware. We use the AFL fuzzer [35] with QEMU to fuzz-test the Linux kernel system call interface, obtaining the same results both in QEMU and on the actual hardware. For the Communication Management Unit (CMU), we use a high-fidelity QEMU implementation of the system, including its most important peripherals for comparison. We produced this implementation by manually reverse-engineering the CMU as our reference. We use AFL to fuzz-test the system call interface of the underlying OS on both the reference and synthesized implementations to confirm we observe the same behavior on both. Although finding vulnerabilities was not the goal of this testing, we nevertheless identified a previously unknown privilege escalation vulnerability in the VRTX kernel used by the CMU (Section 6.4.6).[1]

In summary, the main contributions of this paper are:

- Jetset: a tool for inferring the expected behavior of peripheral devices in an embedded system and synthesizing an executable model of the device (§3).
- Guided symbolic execution using a search strategy based on incremental control-flow graph construction (§4).
- An open source [1] implementation of Jetset using angr and QEMU (§5).
- A demonstration of the general applicability of the Jetset system on thirteen embedded system targets spanning three architectures: x86, ARM, and Motorola ColdFire (§6).
- A demonstration of how Jetset's synthesized devices were used to discover a previously unknown hardware-reproducible privilege escalation vulnerability in the VRTX kernel of the Collins CMU-900 Communications Management Unit (§6.4.6).

[1] Note that this vulnerability is primarily of academic interest, as it is not remotely exploitable and the CMU, while the conduit for digital messages to and from the cockpit, is not considered safety critical for flight.

2 Related Work

Due to the complex nature of firmware and the heterogeneity of the hardware it interacts with, security testing and analysis of firmware is a difficult problem [27, 33]. Different techniques to test and analyze firmware vary both in their goal (e.g., finding bugs, full rehosting, or partial rehosting), as well as the assumptions that they make about the firmware they analyze. For example, a testing technique may only analyze firmware using a particular operating system [7], or may assume that auxillary information about the firmware is available (e.g., firmware-hardware I/O traces [22]) to improve results. The use case of Jetset—partial rehosting using only the firmware itself and no auxillary information—is most similar to other rehosting techniques, however, for completeness, we outline other approaches to analyzing firmware below.

Firmware testing and analysis. Approaches have been developed to test firmware without attempting to create a standalone emulator for the hardware.

Symdrive [30] is a symbolic testing framework for Linux device drivers. Symdrive takes as input the C code for the Linux drivers and attempts to find program paths that violate user written assertions. Symdrive is able to uncover numerous bugs in Linux device drivers; however, it requires source code and is Linux specific.

FIE [15] is a symbolic execution framework that targets firmware for the MSP430 family of microprocessors. FIE takes as input a piece of firmware, a memory map (that denotes which regions are RAM, ROM, MMIO, etc), and an interrupt specification which describes all locations where interrupts could be fired. FIE is designed to analyze all firmware execution paths, which, while effective for the simpler MSP430 microcontroller firmware, is not feasible for more complex firmware like the Raspberry Pi's Linux kernel. For this complex firmware, a more targeted approach (such as Jetset's search strategy described in Section 4) is needed. Furthermore, FIE requires the source code for the firmware—this is how it adds its symbolic execution instrumentation—and it is therefore unsuitable for our needs.

Revnic [8] is a system for symbolically executing driver firmware and reverse engineering its functionality. Revnic takes as input a driver binary, a driver template describing the high level functionality of the driver, and domain specific knowledge about the OS of the driver, and produces source code for the driver. Revnic requires knowledge of the underlying operating system, and requires that the user provide detailed device templates that outline the functionality of the device, and it is therefore unsuitable for the problem of firmware-only emulation.

FirmUSB [23] is a USB-specific symbolic execution framework for analyzing USB microcontroller firmware. FirmUSB takes as input a USB firmware image, and uses domain specific analyses to identify malicious behavior by the USB device. For example, FirmUSB can detect if a device claim-

ing to be a USB keyboard is injecting keys that have not been pressed by looking for USB specific information flows.

Hardware-in-the-loop emulation. Another method of approaching the problem of analyzing firmware is to attach a software emulator running the firmware to the physical hardware, forwarding I/O between the emulator and the firmware.

Avatar [34] is a dynamic analysis framework for embedded systems that takes as input the (possibly instrumented) firmware and the physical hardware, and creates an emulation environment be forwarding I/O between them. Other tools SURROGATES [25] and PROSPECT [24] build on this hardware-in-the-loop approach.

This technique provides the highest fidelity emulation since the emulator directly interacts with the physical hardware; however, use of this technique is contingent on continuous access to the hardware, which is not always possible since hardware (like that used in avionics) may be difficult or impossible to obtain.

Full firmware rehosting. Full rehosting is a technique which attempts to construct a fully featured, high-fidelity emulator from a piece of firmware and auxillary information about the SOC or firmware.

Firmadyne [7] is a platform for automated dynamic analysis of Linux-based embedded systems. Firmadyne takes as input a piece of firmware running the Linux kernel, and executes user-space code for the firmware, emulating the common Linux peripherals. Similarly, Costin et al. [11, 12] extract and rehost the embedded system's filesystem in their own analysis environment to analyze network-facing code. Because the code of interest to an embedded system security analyst is often the user-space, network-facing code, Firmadyne and Costin et al.'s tool are well-suited for this scenario.

Pretender [22] rehosts firmware by recording the interactions between the physical hardware and the firmware. It then uses a machine learning engine to learn a stateful model for peripheral behavior and creates an emulator from this model. Similar to Avatar, Pretender takes as input the firmware, and a connection to the physical hardware, and creates an emulation environment; however, unlike Avatar and related tools, Pretender can fully migrate the firmware to a virtualized environment, and does not require persistent access to the hardware.

HALucinator [10] is a firmware rehosting tool that uses hueristics to locate the code belonging to the hardware abstraction layer (a vendor-provided API for interacting with the hardware) in the firmware and replaces it with manually created handlers. HALucinator takes as input firmware, and the HAL the firmware uses, and produces a fully featured emulation environment for the firmware.

Previous rehosting techniques have relied on auxillary information to infer the behavior of the hardware environment. While this results in a more complete emulator, this auxillary information is not always available—most of our evaluation subjects had none. Furthermore, security analysis is often concerned with only a particular software component of the firmware, (e.g., the network traffic or the file system code) and may not need a fully featured emulator.

Partial rehosting. Partial rehosting, as opposed to full rehosting, attempts to create an emulator from the firmware only, with no auxillary information about the peripherals. However, the emulators produced by partial rehosting are not complete—they are not guaranteed to implement all peripherals for the firmware, only what they can infer. This is the point in the design space that Jetset occupies. There is one other notable system that implements partial rehosting, P²IM.

P²IM [18] does both fuzzing and partial rehosting based on the peripheral model that it infers from the fuzzing stage. It takes as input the target firmware and its memory map, and fuzzes the firmware code by channeling input from an off-the-shelf fuzzer like AFL to the peripherals. It then analyzes the device access patterns exercised during this fuzzing pass to infer details about the MMIO interactions between the firmware and peripheral devices, and executes the firmware without crashing.

There are two key differences between P²IM's fuzzing-based approach, and Jetset's directed symbolic execution-based approach. The first deifference is that unlike P²IM, Jetset is *targeted*—it is designed to ignore most paths through the firmware to focus on a particular target piece of code, which allows it boot deep into large pieces of firmware. While Feng et al. showed P²IM's approach is effective at fuzzing peripheral handling code and emulating microcontroller code, it is not clear whether it scales to larger firmware. Besides evaluating against all of P²IM's publicly available real-world evaluation subjects, we also evaluated Jetset against four complex pieces of firmware—one of our evaluation subjects, the Raspberry Pi 2 is 450x LoC of any of P²IM's evaluation subjects. We attempted to evaluate P2IM on our 4 real-world firmware samples. Unfortunately, the current version of P2IM only supports Cortex-M MCUs and we were unable to run it on any of our samples, including our Cortex-A7 and Cortex-A8 firmware.

The second difference is that, while fuzzing-based approaches are efficient since they use lightweight executions, they can have trouble bypassing complex checks. In Section 6.5.3, we provide an example of a complex numerical check that occurred when inferring the behavior of an FPGA in one of our evaluation subjects. Jetset is able to handle complex numerical checks, because it performs partial rehosting using symbolic execution.

3 Jetset Overview

Jetset uses symbolic execution to infer how peripheral devices must respond to reads from the firmware for execution to progress toward the goal address. It uses this inferred information to deduce and reproduce expected peripheral device functionality to boot firmware in an emulator such as QEMU. This allows analysts to boot the system in an emulator with only the firmware, and without the target's hardware

or support for the peripheral devices in the emulator. To do this, Jetset requires the following information about the target embedded system.

- The **executable code** of the target, usually read out of program flash or extracted from a firmware update provided by a manufacturer.
- The **memory layout** of the target, specifying which regions of the address space are mapped to program memory, RAM, and device I/O registers. This information can be obtained from the datasheet of a single-chip system or from a basic analysis of the executable code. Note that Jetset does *not* need to know which devices are mapped where, only the address range used for MMIO.
- The **entry point address** where execution begins. This is often specified in the CPU datasheet.
- The program **goal address** that the analyst wants the program to reach. For example, this can be the address of a print instruction that reports a successful system boot.

There are two stages of Jetset operation: *peripheral inference* and *peripheral synthesis*. In the inference stage, Jetset uses symbolic execution to infer expected device behavior. Then, in the synthesis stage, the output of the inference stage is used to create a device suitable for use in an emulator (e.g., QEMU).

3.1 Peripheral inference

In the peripheral inference stage, Jetset symbolically executes the firmware to infer what values should be returned by reads from device registers in order for execution to reach the firmware's goal address.

Symbolic execution is a general program analysis technique in which a program is executed while values of interest are kept symbolic, that is, treated as if they could take any value. In Jetset, input from devices is kept symbolic. When a symbolic input–dependent branch instruction is processed, both outcomes are explored. For example, given the statement `if x > 5 then a else b`, both the path starting with statement `a` in which $x > 5$ and the path starting with statement `b` in which $x \leq 5$ will be explored.

3.1.1 Inferring device I/O constraints

Jetset executes the target code in a custom symbolic execution environment. During execution, all reads from MMIO address space are symbolic, while the initial contents of flash and memory are concrete. Each read from a MMIO address returns a distinct symbolic variable; that is, two reads from the same address result in two *different* symbolic values. Using symbolic execution, Jetset can explore all program paths in the firmware that depend on device behavior. Jetset stops when an execution path reaches the goal address, resulting in a set of constraints on values read from device registers that lead to this address.

3.1.2 Searching for the target

The purpose of symbolic execution in the peripheral inference stage is to find an execution path that reaches the goal address specified by the analyst. However, naive forward symbolic execution on firmware of non-trivial size quickly becomes impractical because of the large number of paths that need to be explored. To remedy this, Jetset uses *guided symbolic execution* to favor exploring the most promising paths first. Specifically, Jetset uses the control flow graph (CFG) of the target program to annotate each basic block with a distance to the goal, calculated as the number of CFG edges between the block in question and the block containing the goal address. At a branch, Jetset chooses to explore the basic block with the lower distance to the goal. A search path terminates either when it reaches the goal, triggers a system reset, or enters an infinite loop. To detect infinite loops, Jetset checks against a set of simple infinite loop patterns at the CFG level (see §4.4).

Static CFG generation cannot always recover indirect control flow transfers (e.g., indirect function calls from a function pointer). Because of this, a path to the goal may not be visible in the generated CFG. In this case, Jetset explores paths until it reaches an indirect jump, resolves the jump, and then generates more of the CFG (see §5.1.3).

Jetset calculates a calling context sensitive distance function over the interprocedural CFG to guide its search (see §4.2). A calling context sensitive distance function is one that only includes paths that follow a valid call chain, i.e. all calls that are returned from are returned to the correct location. This distance function is defined to ensure that forward progress in the firmware is being made, and to guide Jetset's search towards the most efficient path to the boot sequence.

While executing, the firmware may require interrupts to be serviced to reach the target.

3.1.3 Injecting interrupts

Because booting the firmware may require interrupts, Jetset periodically injects interrupts during the inference stage. For example, the goal address for the Raspberry Pi firmware is in a different kernel thread than the entry point, so a scheduler interrupt is needed to reach the goal. From Jetset's point of view, this means that it needs to execute an interrupt service routine (ISR) to make progress. Given infinite compute resources, Jetset could explore every possible interrupt either firing or not after each instruction. However, this is impractical. Jetset exploits the fact that well-designed systems are not sensitive to the *exact* timing of interrupts and that ISRs are written to handle spurious interrupts gracefully. Jetset periodically injects interrupts during symbolic execution, so that each ISR is executed periodically during each execution path. If the main execution thread happens to be waiting for an ISR to update a variable, Jetset will eventually execute that ISR, and the thread can continue making progress.

Once Jetset has reached the target (with or without interrupts), it can create a synthetic peripheral model that can be used in QEMU.

3.2 Peripheral synthesis

The result of the peripheral inference stage is a set of constraints on values read from peripherals needed for the firmware to boot. Jetset then uses Z3 [16], the default SMT solver used by angr, to find an instance satisfying these constraints, resulting in a set of concrete values that can be returned in response to device reads during execution. This allows Jetset to construct a light-weight, concrete device model, rendering peripheral inference a one-time cost per device.

3.2.1 Synthesizing an emulator from I/O traces

In effect, the synthesis stage generates an I/O trace that is sufficient to reach the goal in the emulator. The synthesized trace is partitioned by I/O address, so there is a separate trace for each MMIO address. When Jetset reaches the end of an I/O trace for a particular address, any subsequent reads return the last value in the trace. This allows Jetset to continue past the goal address in emulation, but precludes any complex interaction with the device after the trace has ended (see Section 7). After the trace has ended, it is already past the complexities of the initialization stage, and this model is sufficient to carry out useful dynamic analysis tasks on the target firmware (see §6).

The synthetic device also injects interrupts during emulation.

3.2.2 Driving interrupts during emulation

The synthesized device injects interrupts in the same way as during peripheral inference, ensuring that any necessary ISRs are executed in emulation. Interrupt timing during execution in an emulator does not need to precisely match the timing during peripheral inference—if, during emulation, an interrupt is fired one instruction later, this will not make a difference in emulation.

4 Search Strategy

Firmware binaries are too complex to evaluate all possible paths within them. Our interest, though, is in reaching a particular security-critical point deep in the code. Jetset uses a novel application of Tabu search [19] to find a path to the goal address in the firmware. Tabu search is a search algorithm that has been used for searching complex nonlinear search spaces since the 1970's [20]. Jetset uses Tabu search as it allows it to encode domain specific information to improve both path prioritization (Jetset uses a distance function based on the firmware's control flow graph) and backtracking (Jetset uses specialized backtracking rules to avoid failure conditions like hanging firmware).

4.1 Tabu search

Tabu Search is a variation of depth-first search guided by a distance function—it remains on the same path, selecting the closest option at each decision point, until hitting a termination condition. Tabu Search also encodes a *Tabu List* which acts as a blocklist for known bad states, acting as a filter for which states Jetset may backtrack to in the future. In particular, Jetset does not backtrack to states to a location—encoded as a (pc, callstack) pair—that it has already visited. Tabu Search can also encode details such as backtracking strategies and termination conditions, as we elaborate on in Section 4.4. Jetset's search is guided by a context-sensitive distance function.

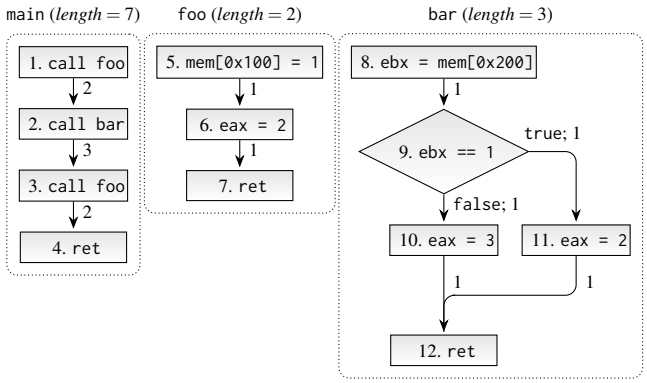

Figure 1: Context-sensitive distance from statement 5 (in first `foo` call) to statement 7 (of second `foo` call).

4.2 Context-sensitive distance

To ensure that Jetset continues to make forward progress towards the goal address, Jetset uses a distance function to guide its search. This distance function is context-sensitive [26]: it takes into account that the distance between two instructions in a program can depend on the calling contexts (i.e., the callstack) of the two instructions. Computing a context-sensitive distance function is more complicated than computing a local distance (i.e., the distance between two instructions in a single function).

The local distance between two instructions is simply the graph distance between the two instructions in the control flow graph. For example, in Figure 1, the distance from statement 5 to statement 7 (both within `foo`) is 2. When computing local distances, the edges for call instructions need to be weighted based on the called function's *length*—the distance between the start of the called function and the nearest return of that function. For example, in Figure 1, the distance from statement 1 to statement 4 is not 3, but 7. This is because, when executing a call instruction, it is not really one instruction being executed, but every instruction until the call returns. This is further complicated, because the called function may itself call other functions. Therefore, to compute local distances for each function, Jetset first creates a callgraph of all functions in the firmware, then computes local distances for functions in topographical order. This ensures that when Jetset computes local distances for a function, it has already computed

local distances for every function that function calls. But this still only gives local distances—it does not provide distances between instructions in different functions.

Computing the distances between instructions in different functions is more complicated, because functions are often called in more than one context and Jetset is only interested in *realizable paths*—paths which follow a valid call-return sequence. For example, in Figure 1, the distance between statement 5 (in `foo`) and statement 4 (in `main`) depends on `foo`'s calling context: if `foo` was called from statement 1, then the distance is 7, if `foo` was called from statement 3, then the distance is 2.

Jetset uses a context-sensitive distance function: it determines the distance between an instruction in one calling context—a (pc, callstack) pair—to another instruction, in another calling context. To compute this distance function, Jetset first precomputes local distances for all functions. Then, Jetset computes the distances between instructions in different functions. To do this, Jetset takes advantage of the fact that all paths between instructions can be broken up into a sequence of returns, followed by a sequence of calls [26] (there will never be an interleaved call and return, because then that would be a local distance!). Nonlocal distances can therefore be seperated into two distances: the *callstack distance*—the distance along the sequence of returns up the callstack—and the *callchain distance*—the distance along the sequence of calls that lead to the goal address (or the goal address in a specific calling context).

Jetset precomputes all local distances, but both the callstack and callchain distances are computed lazily from the actual stack during execution (it is infeasible to precompute all callstack and callchain distances). Jetset computes the total context-sensitive distance as the sum of the callstack and callchain distances.

Callstack distance. The callstack distance measures the distance from an instruction in one calling context to an instruction that can be reached by a sequence of returns (i.e., instructions in functions in the current callstack). To compute the callstack distance, Jetset first computes the local distance to the location of the closest `return` instruction. It continues summing the distances to each return of each function recursively up the call stack. It stops once it reaches a function that can reach the target with a set of calls (i.e., a function that is in the target's callstack), as shown below.

```
# Calculate callstack distance
while function not in target_callstack:
  distance += local_distance(cur, ret)
  cur = function.returns_to
  function = stack.next_function
```

For example, suppose Jetset wanted to reach statement 7 (in the second `foo` call) from statement 5 (in the first `foo` call). The call stack distance would be 2, as that is the distance to exit from `foo` to `main`, at which point statement 7 can be reached by a set of calls.

Callchain distance. The callchain distance measures the distance from an instruction in one calling context to an instruction that can be reached by a sequence of calls. To compute the callchain distance, Jetset first computes the local distance to the nearest `call` instruction that leads to the target. It then recursively sums the distance to each function call on the way to the target, as shown below.

```
# Calculate callchain distance
while function != target_function:
  call = target_callstack.closest_call
  distance += local_distance(cur, call)
  function = call.target
  cur = function.entry
```

Suppose again that Jetset wanted to reach statement 7 (in the second `foo` call) from statement 5 (in the first `foo` call). The call chain distance would be 5: 3 to reach the second `foo` call and 2 to descend into the `foo` call to reach statement 7.

Fallback distance function. In cases where the current interprocedural control flow graph does not contain a path to the goal address, Jetset relies on using the local distance to the nearest return as a fallback distance function. The incremental CFG generation improves the quality of the CFG over time, so eventually the CFG will contain a path to the target.

4.3 Alternating decisions to aid exploration

Jetset's distance function is only an approximation of the real distance—it represents the graph distance on the control flow graph, not the number of blocks that need to be executed to reach the target. Using CFG distance as a heuristic is a powerful technique for guiding execution, but it is only a heuristic—there are situations in which the longer path is the correct path. Therefore, Jetset needs to balance how often it conforms to the distance heuristic, and how often it explores choices that do not follow the heuristic. To do this, Jetset uses a deterministic method to strike this balance between exploring new choices and exploiting the heuristic. It uses an alternation threshold n—every n times Jetset visits a location, Jetset chooses a suboptimal decision. In practice, we find an alternation threshold of three has the best performance.

4.4 Backtracking to avoid error states

Jetset's search algorithm may guide it to a point in the program where it becomes infeasible to reach the target and it needs to terminate the current path and backtrack to a previous state. There are two different cases where this occurs. The first case where Jetset backtracks is when a system reset occurs; it is unlikely that a system reset takes place in a correct boot sequence, and backtracking on system resets allows Jetset to avoid boot loops. The second case where Jetset backtracks is when Jetset enters a statically-detectable infinite loop.

A statically-detectable infinite loop is one where, even if all paths in it were satisfiable, there would still be no exit. These statically detectable infinite loops are efficiently detectable on a control flow graph; for example, there is no way to escape a single basic block that unconditionally branches onto itself. Jetset marks all such points with breakpoints, and upon reaching one, it backtracks. While other work has attempted to efficiently detect infinite loops at runtime [5], this is not a well studied problem at the binary level, and all infinite loops in the boot sequences we have encountered have been statically detectable.

When Jetset backtracks, it backtracks to the last untaken symbolic branch that is closest to the target. If multiple branch-decisions are equally close to the target, than the most recent one is selected. Decisions are identified in a context sensitive manner, so if a particular decision has been chosen under one calling context but not another, then Jetset may still backtrack to that decision under the second calling context.

5 Jetset Implementation

Jetset uses symbolic execution to infer the expected behavior of peripheral devices used by firmware and then emulates this firmware using the resulting synthetic devices. To do symbolic execution, Jetset uses angr [31] and the Z3 SMT solver [16]. To emulate the firmware, Jetset uses QEMU [4]. We modified angr by adding a lifter for m68k/Coldfire as well as adding additional support for privileged x86 code and x86 memory segmentation. We implemented Jetset in 5500 lines of Python code, including changes to angr, and 2000 lines of C. The remainder of this section covers the in-depth implementation details of Jetset's symbolic execution environment and peripheral synthesis.

5.1 Symbolic execution environment

Angr is a symbolic execution engine and general binary analysis platform which provides binary lifting, static analysis, and symbolic execution. We replaced angr's builtin dynamic symbolic execution system with a custom system based on QEMU, allowing Jetset's symbolic execution to be more closely coupled with the underlying hardware emulation environment. Jetset uses angr to generate and analyze the control flow graph, and to manage constraints for symbolic execution.

5.1.1 Whole-system symbolic execution

Jetset does whole-system symbolic execution [9]—it symbolically executes inside of QEMU's full system emulation mode to closely couple the symbolic execution and the hardware environment. Executing directly inside QEMU was also critical to performance. Another benefit of this is that Jetset does not need to encode any complex semantics of threading or interrupts since it uses QEMU's CPU model, which models the delivery of hardware interrupts. Jetset only needs to invoke QEMU's builtin interrupt injection mechanisms.

5.1.2 Interrupt injection

To reach the goal address within a piece of firmware, Jetset may need to invoke an interrupt service routine (ISR). For example, a flag in RAM may need to be set by a particular ISR to proceed further in the boot sequence. Without injecting the interrupt, the main execution thread would otherwise busy loop, waiting for the flag to change,[2] but by injecting the interrupt, the booting process can proceed.

During symbolic execution, Jetset periodically injects interrupts. Spurious interrupts should not cause erroneous behavior in well designed interrupt handling code, so Jetset errs on the side of overapproximation of fired interrupts (i.e., firing more interrupts than are likely to be used in the actual boot sequence). To prevent the boot sequence from hanging while waiting for an interrupt, Jetset injects interrupts in a cycle from 0x1 to the maximum number of interrupts on the architecture. Jetset uses the QEMU builtin `qemu_set_irq` function to trigger interrupts.

5.1.3 Incremental CFG construction

Jetset relies on its distance function to guide execution, but it is not always able to statically generate a complete interprocedural control flow graph. This may occur if, for example, the goal address is on the other side of a virtual function call or an x86 hardware task switch. When the CFG is incomplete, it causes Jetset's distance function to be imperfect, as it may miss shorter paths, or not find a path to the target at all.

To efficiently search for the goal address in the presence of these obstacles, Jetset uses *incremental CFG generation*. If, in the course of symbolic execution, Jetset encounters a symbolic branch in a function that is not in the CFG, it traverses the call stack, adding each function it has not yet seen, and adding edges to the callgraph showing the relationships between these functions. Jetset then recalculates its distance functions over the new control flow graph, an inexpensive computation.

This allows Jetset to improve the accuracy of its distance function over the course of its search. Incremental CFG generation was integral to reaching the goal address in each of our experiments. Checking at each instruction whether Jetset's current location is in the callgraph would be overly expensive, so Jetset only checks for inclusion in the CFG and updates the CFG at each point in which it is making a decision about a symbolic fork. This greatly reduces the cost of incremental CFG generation, and does not reduce the efficacy, as symbolic branches are the only times Jetset uses the distance function.

Even if Jetset can make full use of the distance function and make all the right decisions, symbolic variables (and the constraints on them), can quickly get prohibitively numerous.

[2] Since the flag is in RAM, Jetset cannot infer that the flag should be symbolic.

5.1.4 Optimizing SMT constraints

Jetset uses two optimizations to reduce the number of symbolic variables it has to process during symbolic execution: constraint independence optimization and decision finalization.

Jetset uses constraint independence optimization [6] to reduce the number of constraints used when checking satisfiability of paths during symbolic execution. Before checking the satisfiability of the current path, Jetset only submits the constraints used in the current symbolic branch to the SMT solver, vastly reducing the number of symbolic variables processed each branch. Jetset records constraint independence sets—which symbolic variables are dependent on what other symbolic variables—efficiently with a disjoint set data structure.

For very complex firmware (like the Raspberry Pi 2) Jetset also uses decision finalization—after branching on a symbolic value at the same address `n` times without crashing, Jetset stops symbolically handling that variable, and makes it concrete before the synthesis stage. In general, Jetset keeps all device reads fully symbolic, but after a sufficient number of checks (in Jetset's case, 200) on a device read at the same location without crashing, it is unlikely that returning the same value would cause the firmware to crash. By concretizing the device read early, decision finalization reduces the number of symbolic variables needed for complex firmware.

5.2 Peripheral synthesis

The output on the inference stage is a set of constraints on values read from device registers along a path from the entry point to the goal. In the synthesis stage, Jetset generates a synthetic device that satisfies these constraints. In response to each device read, the device returns the concrete value that will guide the execution down the path from the entry point to the goal. When this device is used with a concrete emulator (like an unmodified QEMU), the firmware will boot to the goal address.

5.2.1 I/O synthesis

Memory-mapped I/O is the primary mechanism by which firmware interacts with hardware devices. Firmware often makes decisions based on the results of this I/O. For example, during the hardware initialization phase of the firmware, the firmware checks if devices are present and working properly by writing to the device and expecting a particular status flag to be returned when reading from the device. To emulate these reads, Jetset treats memory mapped I/O regions as symbolic data. MMIO regions differ from standard symbolic memory locations, though, because two consecutive reads from a memory mapped I/O region may not return the same value. For example, firmware may read from a timer data register, then continue to read until the value read from the register changes. Jetset models this behavior by having each read return a different symbolic variable.

Jetset tracks the constraints placed on values read from memory mapped I/O regions so that when it generates a synthetic device, it can ensure that the synthesized values conform to these constraints such that the path discovered during symbolic execution is taken during emulation.

Once Jetset finds a path that boots the firmware, it no longer needs to perform any inference. It then generates a device emulator that can be used with an unmodified QEMU instance to avoid complexity during subsequent dynamic analyses and to avoid the overhead that symbolic execution incurs. To generate the device emulator, Jetset starts by extracting the symbolic device I/O trace from the successfully booting program path. It then concretizes the values of all I/O reads under the set of constraints that led to successfully finding the boot address. This concrete I/O trace is partitioned by memory address to create by-address I/O traces. These traces are treated as read queues, so that when the synthetic device is read from, the appropriate read queue is accessed, and the synthetic device responds with that value. If the queue is emptied (i.e., execution is beyond the intended boot address), the device responds with the last value of the queue.

Jetset then outputs a C file that implements this device read handler as a properly formatted QEMU device emulator file. This device emulator is then added as a device to an unmodified distribution of QEMU. When QEMU runs the firmware with this device, it will boot to the intended boot address, at which point we can perform dynamic analysis. The device models Jetset produces only replay one possible boot path, and, after that, replay the last MMIO value that allowed the firmware to progress for the address being read from. The intuition behind this model is that each MMIO address is likely being used for one purpose, for example, a status ready flag, or a configuration variable. In the first case, we always want to return the status flag that allows the firmware to stop polling, which should be the value returned by this simple device model. In the second case, configuration values are not often changed after initial configuration, and if they are, they initial configuration is still valid and does not affect the behavior of the portion of firmware under analysis. Therefore the simple model continues to return the correct configuration variable as it returns the last known value of the configuration variable that satisfies the constraints of the boot path. In Section 6, we show our synthesized model faithfully emulates the systems under test.

But even if the synthesized device emulates all I/O correctly, it still needs to reproduce the interrupts used in the synthesis stage as well.

5.2.2 Interrupt synthesis

Jetset injects interrupts during concrete emulation to ensure that the synthesized emulator follows the same path to the program goal as the Inference stage. To follow the same path as closely as possible, it injects interrupts using the same interrupt strategy as during the Inference stage, that is, it cycles through all possible interrupts in the same order. Although

this does not guarantee that the interrupts are injected in the exact same location, i.e., between the same two instructions, it does preserve the order and relative frequency of interrupts. We rely on the same assumption made during the Inference stage—that interrupt handling code does not rely on highly precise timing of interrupts, and that it should handle spurious interrupts gracefully. We found this assumption to hold when we evaluated Jetset on real firmware targets.

6 Evaluation

To evaluate Jetset, we use it to infer peripherals for thirteen embedded systems. Nine of these systems are systems evaluated by the P²IM [18], and four are original targets (Table 1). We then synthesize the peripherals and use them to boot the target system firmware in QEMU. We chose the nine P²IM subjects since they represent a wide range of use cases, and use a variety of MCUs, peripherals, and operating systems. We chose the CMU-900 and SEL-751 because their security analysis was of independent interest to us. The other two, a Raspberry Pi 2 and BeagleBoard-xM, represent widely used SOCs. In the case of the Raspberry Pi 2, it also allowed us to compare the fidelity of our emulation to the actual hardware system.

Our evaluation aims to determine whether using symbolic execution to infer expected peripheral device behavior works well enough to be useful. We do this in two ways. First, we synthesize the inferred peripheral device models and instantiate them in QEMU. We then execute the system code in this QEMU instance (with inferred device models) to determine whether the system will boot to the goal address we targeted in the Inference stage. This tells us that the synthesized devices mimic the expected peripherals well enough for the system to get to the intended target address in the code, at which point it is ready for further dynamic analysis.

For two of our targets, the CMU-900 and Raspberry Pi 2, we go further and use the booted system to fuzz-test the system call interface of both operating systems, VRTX and Linux, respectively. This end-to-end test allows us to compare the behavior of the emulated system to a reference, to determine if the emulated system is a good stand-in for the original. For the CMU-900, our reference is an instance of the system running in QEMU using models of the peripherals based on painstaking manual reverse-engineering. For the Raspberry Pi 2, we do the same fuzz-testing using the hardware board itself, and confirm that the results of system call fuzzing are the same in both. Although it is not the goal of this work to identify specific vulnerabilities, we did find a number of crashes in the CMU-900's operating system kernel. One of these crashes was manually adapted into a privilege escalation exploit. Because our testing targeted the system call interface, absent an additional vulnerability, this bug is *not* remotely exploitable. We did not identify any exploitable privilege escalation vulnerabilities in the Linux kernel (nor did we expect to find any); instead, our tests confirmed that both systems responded to fuzzer input *in the same way*.

We begin our evaluation by describing our methodology in Section 6.1. In Sections 6.2 through 6.5, we describe the results for our four original targets in detail. Finally, in Section 6.6, we describe our results for the P²IM targets.

6.1 Methodology

To evaluate Jetset's performance on each of our targets, we begin by collecting the necessary information about each target. As described in Section 3, this information consists of the executable code of the system, its memory layout, the entry point where execution is to start, and the goal program address that we would like to reach. Using the information above, we configured and ran Jetset to infer peripheral device behavior and then used the QEMU device synthesis module of Jetset to generate executable C models of the devices identified. We ran all experiments on an Intel Xeon Silver 4208, 2.10 GHz, 32-core server running Ubuntu 18.04.3 LTS. We then used these synthetic peripheral device models to boot the firmware image in QEMU and ensured they reached the expected goal.

Table 1 reports the statistics about each stage for our four original targets. References to blocks in the table (e.g., *Blocks executed on path*) are to program basic blocks, the basic unit of translation in QEMU. Each target is discussed in detail below.

6.2 Target: Raspberry Pi 2

The Raspberry Pi 2 is a single-board computer based on the Broadcom BCM2836 system-on-a-chip (SoC). The BCM2836 has a quad-core ARM Cortex-A7 processor, a Broadcom VideoCore IV 3D GPU [28, 29], and numerous peripherals. The Raspberry Pi 2 runs a modified Linux kernel that includes binary drivers for some of the devices on the BCM2836 SoC. On the Raspberry Pi 2 hardware, the first stage bootloader is executed by the GPU, which loads a device tree blob[3] and the Linux kernel into memory, and then transfers control to the kernel. (There is no publicly-available emulator for the VideoCore GPU, and the GPU boot code is provided in binary form only.)

The current version of QEMU supports the Raspberry Pi 2, implementing 13 of the 28 peripherals defined in device tree blob file included with the official Linux kernel from the Raspberry Pi Foundation. The remaining 15 devices are unimplemented; reads from their device registers always return 0. In addition, QEMU does not emulate the VideoCore boot: instead, it loads the device tree block and kernel into the RAM device directly, and then transfers control directly to the kernel. In our evaluation, we use both the hardware Raspberry Pi 2 and the QEMU-emulated Raspberry Pi 2 to test the fidelity of our emulation.

[3]The device tree blob is a compact description of the hardware configuration used by the operating system kernel to locate peripheral devices [17].

Table 1: Evaluation targets and summary statistics.

	Raspberry Pi 2	BeagleBoard-xM	CMU-900	SEL-751
CPU/SoC	Broadcom BCM2836 (ARM)	TI DM3730 (ARM)	AMD Am486 (i386)	NXP MCF54455 (ColdFire)
OS/SW	Linux 4.19.y	X-Loader	VRTX-32	G5.1.5.0
Peripheral inference				
Wall-clock time	6m43s	5m15s	5m20s	2h34m51s
Blocks in code base	238,792	872	55,016	141,750
Total blocks executed	81,194,393	20,198,824	53,143,508	3,351,484,857
Blocks executed on path	81,194,393	20,198,824	27,517,932	3,351,484,857
Unique blocks executed	43,157	484	776	11,364
Unique blocks executed on path	43,157	484	731	11,364
MMIO writes (ignored) on path	84,060	938	1,308	32,480
MMIO reads (symbolic) on path	83,857	3,633	242	704
MMIO write addresses on path	40	244	13	68
MMIO read addresses on path	37	61	5	26
Devices accessed	6	11	5	5
Peripheral synthesis				
Wall-clock time	3.16s	5.64s	0.018s	5.61s
Total Symbolic Variables	1,384	3,633	242	704
Total Constraints	5,226	8,353	756	11,142
Constraints per variable	3.78	2.30	3.12	15.83
Average trace length	37.4	59.56	48.4	27.08
Median trace length	1	3	5	2
Maximum trace length	1076	2,770	215	343
Emulator execution to goal				
Wall-clock time	8s	101ms	289ms	1m1s
Total blocks executed	81,454,594	20,198,656	27,519,080	3,351,502,947
Unique blocks executed	43,255	483	731	11,364
MMIO writes (ignored)	83,915	938	1,882	32,480
MMIO reads	83,857	3,633	242	704
MMIO write addresses	43	244	13	68
MMIO read addresses	27	61	5	26
Devices accessed	6	11	5	5

6.2.1 Raspberry Pi 2 configuration

We took the official QEMU Raspberry Pi 2 configuration as our starting point. Specifically, we used the same MMIO address ranges as the official QEMU configuration. Of the 13 Raspberry Pi 2 peripherals implemented by QEMU, we kept three devices that implement part of the VideoCore IV, leaving the remaining to be inferred by Jetset. The VideoCore IV is used to perform DMA to RAM, and we do not currently attempt to infer DMA behavior. Generally, targets that rely on DMA writes to RAM would need support for the DMA controller in the emulator. Our target code was an unmodified Raspberry Pi 2 kernel with a stub init process (instead of the original initramfs), which we used to drive our kernel system call fuzzing.

We chose the `run_init_process` function as our goal program address. This function is invoked to transfer control to the `init` process for the first time. Program execution reaching this function indicates that the kernel boot sequence has finished.

6.2.2 Inferring the Raspberry Pi 2 peripherals

The peripheral inference stage completed in under 7 minutes after executing 81.1 million basic blocks. Table 1 summarizes this, and other parts, of the evaluation. Jetset did not backtrack during execution (total blocks executed equal blocks executed on path). This is because our distance function (Section 4.2) avoids execution paths that would result in backtracking (i.e. an infinite loop or halt). This does not mean that Jetset finds the *shortest* possible path to the goal; rather, the distance function helps it avoid terminating paths.

Table 1 also shows that the number of blocks executed is over three orders of magnitude greater than the total number of blocks because of loops that execute a set of blocks repeatedly. In most cases, these loops operate on concrete values only, allowing QEMU to execute them efficiently.

In all, on the path from entry point to goal, Jetset saw 84,060 MMIO writes to 40 distinct write addresses and 83,857 MMIO reads from 37 distinct read addresses, covering 6 of the 28 devices defined in the device tree blob.

6.2.3 Synthesizing the Raspberry Pi 2 peripherals

The 83,857 MMIO reads together introduced 1,384 symbolic variables, with an average of 5,226 constraints per variable. This disparity between the number of reads and number of symbolic variables is caused by Jetset's decision finalization optimization (see Section 5.1.4). Decision finalization was effective in this case because there were many repeated reads from the same addresses in the Raspberry Pi 2 boot process—one example being reads from the serial interface's status register.

Jetset synthesized the synthetic device in less than four seconds. The median trace length was 1, whereas the longest was 1076 values from a UART status register. Table 3 (in the Appendix) shows part of the synthesized I/O traces.

The Raspberry Pi 2 kernel interacts with several devices, such as the random number generator and custom SD card host controller, for which no public documentation exists. Nevertheless, Jetset was able to infer, from the driver code that interacts with these devices, what values these otherwise opaque devices need to produce in order for the system to boot.

6.2.4 Emulating the Raspberry Pi 2

We configured QEMU to use our synthesized peripherals and booted the same kernel used in the inference stage. The kernel reached the `run_init_process` function in 8 seconds. Table 1 summarizes the statistics of emulator execution using synthetic devices.

One significant difference between the execution trace from the peripheral inference stage and execution in the emulator with synthesized devices was in the behavior of the SD host controller. Specifically, slower inference-stage execution led to controller command timeout, resulting in an error message and a register dump. However, during emulated execution with synthetic devices, the SD host controller initialized without a command timeout. Jetset is resilient to timing differences because Jetset partitions I/O traces by address. Thus, the relative order of reads from the same MMIO address will remain the same, even if peripheral devices are accessed in a different order during inference and emulation.

Nevertheless, while in this case this timing difference did not prevent the system from booting, such divergence is undesirable, as it may take the emulated execution along a path that ultimately fails. We are currently investigating ways of ensuring tighter timekeeping accuracy between inference and emulation to ensure that the kind of timing differences observed above are less likely to occur.

6.2.5 Further dynamic analysis on the Raspberry Pi 2

With the Raspberry Pi 2, we have both official QEMU support for the target and the target hardware, which allows us to compare the behavior of QEMU using our synthesized peripherals against these two references. After reaching the goal (entry into the `run_init_process` function), we continued execution and used our stub init process to fuzz the kernel system call interface. Fuzzing entails generating random inputs to an interface to elicit unusual, potentially exploitable, behavior. System call fuzzing has been used to find hundreds of bugs and vulnerabilities in commonly used software [36]. We used the AFL fuzzer [35], extended to allow fuzzing I/O peripherals, function parameters, and interrupts. Our fuzzing targeted the Linux system call interface. However, because the Linux kernel is used widely, we did not expect our testing to identify new vulnerabilities in its system call interface. Instead, our goal was to determine whether our synthesized configuration *behaves the same way:* for all three implementations, we monitored the response of the Linux kernel to each system call and recorded which of the following four observable behaviors resulted:

- **Kernel "oops" or panic.** Both indicate a kernel fault, pointing to a potentially exploitable bug. A kernel "oops" does not halt the system, while a panic does.
- **Process killed.** The kernel kills the process issuing the system call. In our configuration, the only process is the init process, leading to a kernel panic with a unique error message. Under normal circumstances process death would not result in a panic.
- **System call return.** The kernel returns to the calling process. We recover any set errno and return values.

In our experiment, we issued 1,571,576 distinct system calls from our init process stub to the Linux kernel running in QEMU with our synthetic devices, resulting in 123,198 unique codepaths. Of these, none resulted in a kernel "oops" or kernel panic. 51,638 resulted in the kernel killing the init process, and 71,560 in the system call returning to our user process. We then carried out the same experiment (using the same exact system calls) using the official QEMU Raspberry Pi 2 configuration with manually-implemented devices. In all 123,198 cases, we observed the same behavior in both the synthetic and manually-implemented configurations, down to fuzzing paths discovered, error stack traces, `errno` values, and system call return values.

To compare our (synthetic) implementation against the real hardware, we selected a random sample of 14,661 test cases. We then booted the Raspberry Pi 2 kernel on the target board and used a custom init process to read system call parameters from a serial port, issue the system call, wait three seconds, and then reboot the system. Using this interface, we issued 14,661 system calls on the target hardware. If the system call returned, our init process printed `errno` and the return value to the serial console. Otherwise, we relied on kernel serial console output to determine whether the init process was killed or whether the kernel encountered a fault ("oops" or panic). We observed the same behavior on the physical hardware as in the emulator with synthetic devices.

6.3 Target: BeagleBoard-xM

The BeagleBoard-xM is a single-board computer based on the Texas Instruments DM3730 SoC [3]. The DM3730 has an ARM Cortex-A8 processor, a DSP processor, a graphics accelerator, and numerous peripherals. The BeagleBoard-xM runs a modified version of the Linux kernel that includes binary-only drivers for the devices on the SoC. Linaro Foundation also provides a QEMU configuration for the BeagleBoard-xM with support for 25 of the 35 peripherals defined by the Technical Reference Manual [32]. The Beagleboard architecture is kernel independent—running the emulated device in QEMU does not require specifying an operating system image.

As our target, we chose X-Loader,[4] the first-stage bootloader on the BeagleBoard-xM, which, in typical usage, is responsible for loading the second-stage bootloader from the SD card. Evaluating the full Linux kernel boot on the BeagleBoard-xM would have required either enabling support for the SD host controller in QEMU to allow the bootloader to load second-stage bootloader and continue the boot sequence, or implementing the direct boot mechanism used by QEMU for the Raspberry Pi 2. Because the BeagleBoard-xM is not of independent interest to us as a target, we chose to target the first-stage bootloader only.

6.3.1 BeagleBoard-xM configuration

We configured QEMU for the ARM 32-bit architecture and defined the program memory, RAM, and MMIO regions as defined in the OMAP35x Technical Reference Manual [32]. We do not include any of the peripherals defined by the manually implemented QEMU configuration. We reused the code provided by Linaro to initialize the CPU and attach RAM memory regions. We chose program address `0x80008000` as our goal, which is the entry point to the second-stage bootloader.

6.3.2 Inferring the BeagleBoard-xM peripherals

In normal use, X-Loader attempts to find a device from which it should load the subsequent-stage bootloader. In particular, it checks for data from the boot sector of the SD card, in nand flash, and via the UART serial port. During this process, it probes 11 peripherals (Table 1) and executes over 20 million program basic blocks, despite the small code base (872 basic blocks). In all, on the path from entry point to goal, Jetset saw 938 MMIO writes to 244 distinct write addresses, the largest number of writes and number of distinct addresses of our four evaluation targets, covering 11 of the 35 devices defined in the device tree blob.

6.3.3 Synthesizing the BeagleBoard-xM peripherals

The 3,633 MMIO reads each introduced a symbolic variable, with and average 8,353 constraints per variable. Jetset then synthesized a distinct I/O trace for each of the MMIO read addresses in less than 6 seconds. The shortest trace was a single value, while the longest was 2,770 values from a UART control register. The path chosen during the inference stage directs the bootloader to boot from the serial port. Jetset infers a fragment of the Kermit file transfer protocol [14] that causes X-Loader to proceed to boot after receiving 0 bytes of the payload.

6.3.4 Emulating the BeagleBoard-xM

We configured QEMU to use our synthesized peripherals and ran X-Loader, as in the inference stage. X-Loader reached the goal address function in 29 seconds. The emulator crashed after reaching our desired boot address while attempting to perform a serial boot. After reading data from serial and writing it to RAM, the firmware attempted to jump to this "code" loaded from our synthetic device. This resulted in a crash, since Jetset has no method to generate valid ARM assembly and providing it to the serial reads.

6.4 Target: CMU-900

The Collins Aerospace CMU-900 is an electronic system used on many Boeing 737 aircraft. It is responsible for handling digital communications between the aircraft and ground stations. The primary processor of the CMU-900 is the AMD Am486 [2], an Intel 486-compatible processor. The CMU-900 peripherals are implemented as discrete ICs (the Am486 is only the CPU) as well as a Intel 386-based I/O processor board. The Am486 accesses some of these peripherals using port-mapped I/O and some using MMIO. For the sake of brevity, we refer to both as MMIO. We extracted the flash memory image from the flash ICs. We reverse-engineered part of the code to determine the coarse memory layout, that is, which address ranges are mapped to flash, RAM, and MMIO. This was facilitated by the designers' choice to use the x86 memory segmentation system, allowing us to recover the necessary address ranges from the global descriptor table set up early in the boot process. Based on the strings found in the flash image, we determined that the system was running VRTX-32, a real-time operating system. In addition to the OS kernel, we identified ten user-space tasks that implement application functionality.

6.4.1 CMU-900 configuration

We configured QEMU to emulate a 486 processor with the memory layout, as noted above. The entry point into the code was the default entry point for the 486, namely address `0xffffff00`. As our goal, we chose the first system call in task 1 (later, we will use this to fuzz the VRTX-32 system call interface). We did not define any peripherals in QEMU.

6.4.2 Inferring the CMU-900 peripherals

Peripheral inference on the CMU-900 took under six minutes after executing 53 million basic blocks. The CMU-900 required extensive backtracking because the panic function in the CMU-900 can return (when called with a non-fatal error argument) but enters an infinite loop when called with a fatal error argument.

[4] https://github.com/joelagnel/x-loader

In all, on the path from entry point to goal, Jetset saw 1,308 MMIO writes to 13 distinct write addresses and 242 MMIO reads from 5 distinct read addresses, covering 5 devices. With the exception of a single 32-bit read, all I/O on the chosen path was port-mapped.

6.4.3 Synthesizing the CMU-900 peripherals

The 242 MMIO reads each introduced a symbolic variable, with an average of 3.12 constraints per variable. Jetset then synthesized a distinct I/O trace for each of the MMIO read addresses in less than six seconds; Table 4 (in the Appendix) shows the synthesized I/O traces. Based on the values read and written to device registers and an examination of the physical board, we were able to determine that the peripherals were a Z85C30 serial controller at `0x2000`, a DS1685 at `0x3000`, the I/O processor board at `0x5000`; we were unable to determine what device was at `0x6000`. The board also has a programmable interrupt controller and programmable interval timer, however the operating system only writes to their registers during the boot sequence, so Jetset did not need to infer any values read from them.

6.4.4 Emulating the CMU-900

We configured QEMU to use our synthesized peripherals and proceeded to boot the same firmware image, reaching the goal in less than a second. The CMU-900 executed 27,519,080 blocks during emulation, 1148 blocks more blocks than during peripheral inference. This was caused by interrupt injection timing differing slightly between code execution during the inference stage and in emulation.

After reaching the CMU-900 goal address, the emulator continues to execute without crashing, looping through active tasks in the scheduler.

6.4.5 Further dynamic analysis on the CMU-900

QEMU implements four of the peripheral devices used by the CMU-900 (the real-time clock, interrupt controller, interval timer, and serial controller). We created a custom QEMU configuration mapping these devices at addresses expected by the code, which allowed us to compare the behavior of the emulated system with our synthetic devices against a QEMU configured with their full implementation. As with the Raspberry Pi 2, we compared the behavior of the two systems by issuing system calls from unprivileged task 1. Specifically, we stop execution immediately before task 1 issues the first system call, and set the contents of registers using values generated by AFL [35].

AFL found 2963 unique crash code paths during 200 hours of fuzzing. To compare the behavior of our two QEMU implementations (synthetic and manual), we compared the debugging console output produced by the CMU-900.[5] In the case of a successful system call return, the CMU-900 continues with its normal unprivileged task startup sequence. In the event of a protection violation, the CMU-900 prints a wealth of debugging information, which we use to determine whether the two configurations behaved similarly.

Of the 2963 execution paths discovered by fuzzing, 2884 (97.3%) code paths exhibited identical behavior. Another 36 (1.2%) had the same outcome, but differed in the values in some of the registers. The remaining 43 (1.5%) also had the same outcome, but differed more extensively in the output generated.

6.4.6 Privilege escalation

Manual analysis of the 2963 execution paths led to the discovery of a privilege escalation vulnerability. The vulnerability occurs because a single byte can be "leaked" from unprivileged code into the offset of a call instruction in the VRTX kernel. One of the 256 potential values for this byte results in the call targeting the middle of a function. From here, a Return-Oriented Program (ROP) chain can be used to modify the global descriptor table (GDT). A few instructions later, the GDT modification causes the kernel protection error handler to fire. However, the GDT modification changes the base address of the data and stack segment used by the handler so they overlap with an unprivileged data segment. The handler includes a far call whose address is dependent upon a read from the corrupted data segment. A malicious address can be given to this far call, leading to a second ROP chain which further modifies the GDT. This ROP chain changes the address range limits for privileged code and transfers control to unprivileged, writable memory while remaining in processor ring 0.

Because we destructively disassembled the CMU to extract the firmware on which we performed this emulated analysis, it was not possible to use it for validation. However, we were able to validate the exploit on another CMU-900 (one with a slightly different part number and memory layout) after some minimal adaption. In particular, due to small changes in the VRTX kernel between device versions, the exploit's control transfer required supplying one ROP gadget address via a segment rather than a data register and changing some gadget offsets.[6]

To our knowledge, none of the discovered crashes can be triggered remotely. Thus, taking advantage of these crashes would require require carefully constructed application code to already be present and running on CMU. As well, the CMU is not directly involved in flight control and is not considered a safety critical system. Nevertheless, in an abundance of caution, we have disclosed this issue to Collins Aerospace. We have worked closely with the company and have provided sufficient technical detail to replicate our findings and incorporate this information into their own internal risk and safety assessments.

[5] We would prefer to compare the synthetic device QEMU instance to the actual hardware. However, unlike the Raspberry Pi 2, we desoldered chips from the CMU to extract firmware, making the device inoperable.

[6] In addition, we also needed to develop a loading and bootstrap capability to introduce our software into the physical CMU, but that had been developed independently as part of a previous project [13].

6.5 Target: SEL-751

The Schweitzer Engineering Laboratories SEL-751 feeder protection relay is used to protect power grid systems. It consists of several boards plugged into a backplane. The main processor is the MCF54455, a 32-bit microprocessor implementing the ColdFire ISA, a derivative of the Motorola 68000. The MCF54455 includes a DMA controller and several peripherals on-chip. In addition, the SEL-751 also has an Altera Cyclone III FPGA on one of the boards plugged into the backplane.

6.5.1 SEL-751 configuration

QEMU already has support for the ColdFire ISA and required minimal changes to support our processor variant. We configured QEMU using the memory layout specified in the MCF54455 Reference Manual [21], designating address range `0xfc000000`–`0xfc100000` for MMIO. In addition to the on-board peripherals, we determined (through reverse-engineering) that the FPGA was mapped to `0x30000000`–`0x30010000`. We also added support the ColdFire architecture to angr, which did not have support for ColdFire or its predecessor, the Motorola 68000.

We bypassed the bootloader and started execution at the entry point to the operating system. We set our goal to the first point at which the timer interrupts were enabled, which enables the schedule to switch to other tasks.

6.5.2 Inferring the SEL-751 peripherals

As shown in Table 1, peripheral inference on the SEL-751 took more than 2.5 hours, considerably longer than on other targets. Our search strategy led execution down a path which, while not requiring backtracking, engaged in a time-consuming memory operation with little impact on synthesis. In all, the SEL-751 read the registers of the FlexBus controller, the Ethernet controller, the I2C communication interface, and the GPIO system.

6.5.3 Synthesizing the SEL-751 peripherals

The synthesis stage took 6 seconds to synthesize the peripherals for the SEL-751, emulating device reads from 26 addresses. Table 5 (in the Appendix) shows part of the synthesized I/O traces. The SEL-751 had an average of 15.83 constraints per variable, higher than the other targets which ranged from 2.30 on the BeagleBoard-xM to 3.78 on the Raspberry Pi 2. Complex operations on values read from the FPGA were the reason for the larger number of constraints.

Jetset's analysis of the SEL-751 included the inference of five 32-bit FPGA reads in a range of $995 \leq x < 10,000$, where x is a linear translation of the read value. During synthesis, Jetset's SMT solver was able to quickly find the correct input values from the collected constraints (Table 1). Performing this inference via fuzzing where hardware read values are picked uniformly at random would have a success probability of approximately 2^{-94}.

Table 2: P²IM targets. These targets span three different operating systems and four different SOCs.

Target	SOC	OS
Robot	STM32F103RB	Bare Metal
PLC	STM32F429ZI	Arduino
Gateway	STM32F103RB	Arduino
Drone	STM32F103RB	Bare Metal
CNC	STM32F429ZI	Bare Metal
Reflow Oven	STM32F103RB	Arduino
Console	MK64FN1M0VLL12	Riot
Steering Control	SAM3X8E	Arduino
Heat Press	SAM3X8E	Arduino

6.5.4 Emulating the SEL-751

We configured QEMU to use our synthesized peripherals and booted the image used in the inference stage. The kernel reached the goal address in just over a minute after executing 3.3 billion blocks. After reaching the boot address, the firmware continues executing without crashing, repeating a communication loop with the FPGA.

6.6 Target: P²IM firmware

We evaluate Jetset on the nine publicly-available real-world pieces of firmware used to evaluate P²IM [18]. These systems use four different ARM SOCs, two Cortex M-3 (STM32F103RB and SAM3X8E), and two Cortex M-4 (STM32F429ZI and MK64FN1M0VLL12) CPUs. Five of these systems use Arduino as their operating system, one uses the RIOT operating system, and three run on bare metal (shown in Table 2).

6.6.1 Firmware configuration

We configured QEMU to use a Cortex M-3 or Cortex M-4 CPU as appropriate, and used the program entry point and initial stack pointer as specified in the firmware's vector table. For each of the P²IM targets, we used the memory layout specified in their SOC's datasheets as their memory specification. We allocated the full region allowed for MMIO as a single contiguous MMIO block, not differentiating between the different devices. We did not use any predefined peripherals from QEMU, all devices were inferred. For each of the P²IM targets, we use the start of an application-specific event loop as the target for Jetset. These event loops are easy to locate: each of these pieces of firmware begin with a device initialization process, followed by a loop that reads information from peripherals, and response to this input. We select these event loops as our targets, as they dictate the program logic of the application, and are therefore most likely to have application specific bugs.

6.6.2 Inferring the firwmare peripherals

Jetset took an average of 59.7 seconds to reach the target in each of the nine P²IM subjects, and executed an average

of 55,739 basic blocks getting there. Jetset performed an average of 892 MMIO reads to an average of 24 distinct addresses, and 354 MMIO writes to an average of 44 distinct write addresses. These MMIO reads and writes accessed an average of 9 distinct devices for each system.

6.6.3 Synthesizing the firmware peripherals

Synthesizing the devices for each of the P²IM systems took an average of 1.7 seconds. The synthesized devices produced an average of 892 synthesized reads for 24 distinct addresses. The variables produced by these reads had an average of 2.6 constraints per variable and resulted in 9 devices being synthesized per subject. These concrete traces executed an average of 44,258 basic blocks.

6.6.4 Emulating the firmware

Each of the concrete devices generated by Jetset reached the target in the firmware. The concrete booting process took an average of .35 seconds. None of the nine subjects crashed after being concretely executed to the boot address. Most of the firmware got booted to a loop in which the firmware read out new commands from the peripherals, and attempted to execute them.

7 Limitations

As we show in Section 6, Jetset works well for firmware running on a variety of embedded system architectures across multiple application domains. However, our current implementation is not without limitations.

Path correctness. Jetset has no knowledge of the underlying hardware other than the behavior that is observable to the CPU. The path taken through firmware is not necessarily one that may ever be returned by the hardware; however, the path taken is one that is acceptable to the firmware—no interaction with any of the peripherals results in a boot failure. While the execution of firmware running on physical hardware is constrained in its behavior by how the physical peripherals really behave, these system constraints are external to the firmware and cannot be inferred without auxiliary information about its behavior.

Limited peripheral model. Jetset does *targeted rehosting* in that it only constructs an emulator that is sufficient to emulate the software component-under-test. If the firmware reads from a peripheral's address after reaching the target, Jetset replays the last satisfying value read from that address. In our tests, we found that this simple model is sufficient to perform useful analysis and bug finding (as shown in Section 6); however, more complex interactions with the peripherals may not be emulated correctly.

Another limitation of Jetset's peripheral model is that Jetset has no understanding of the semantics of the devices synthesized besides what is needed to guide the firmware towards the target address. We found that our limited peripheral model caused the firmware to crash after reaching the target address in one case. In the BeagleBoard-xM, we found that our emulator attempted to execute data loaded from a serial boot from our synthetic device. Jetset had no method to detect that the data it is returning from device reads should be valid ARM code, and crashed because of it.

In future work, we plan to have Jetset synthesize more complex, stateful peripheral models as well as identify known peripherals with existing emulator implementations.

No DMA support. Jetset does not support devices that perform direct memory access (DMA) to normal RAM. This is because DMA is not observable by firmware since the device accesses memory without the assistance of the CPU. In the two cases that DMA was required to boot the firmware-under-test, we either left the DMA device in the QEMU model (as described in 6.2) or manually marked the DMA region symbolic (as in the Robot firmware in 6.6). We leave automated modeling of DMA to future work.

8 Conclusion

We described the design and implementation of Jetset, a system that uses symbolic execution to automatically infer what behavior embedded system firmware expects from its target hardware. We use this inferred behavior to synthesize models of target hardware devices that can be used to execute the firmware in an emulator. We demonstrate that the Jetset technique is general by evaluating it on multiple computer architectures, operating systems, and application domains.

The inferred device models allowed us to boot target firmware in an emulator (QEMU), saving considerable engineering effort that would be required to reverse-engineer the hardware. Once booted to the specified target, an analyst can then perform variety of dynamic testing tasks on the code of interest. We demonstrated one such task: fuzz-testing system calls on firmware from a Boeing 737 avionics system.

9 Acknowledgements

This material is based upon work supported by National Science Foundation awards CNS-1646493 and CNS-1901728, and DARPA award FA8750-16-C-0181.

References

[1] Jetset. `https://jetset.aerosec.org`, 2021.

[2] *Enhanced Am486DX Microprocessor Family*. Advanced Micro Device, Inc., March 1997.

[3] *BeagleBoard-xM Rev C System Reference Manual*. BeagleBoard.org Foundation, April 2010. Revision 1.0.

[4] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of USENIX Annual Technical Conference 2005*, pages 41–46, April 2005.

[5] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of ASE 2009*, pages 161–169, November 2009.

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2), December 2008.

[7] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of NDSS 2016*, February 2016.

[8] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of EuroSys 2010*, pages 167–180, April 2010.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S^2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems*, 30(1), February 2012.

[10] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of USENIX Security 2020*, August 2020. To appear.

[11] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proceedings of USENIX Security 2014*, pages 95–110, August 2014.

[12] Andrei Costin, Apostolis Zarras, and Aurélien 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*, pages 437–448, 2016.

[13] Sam Crow, Brown Farinholt, Brian Johannesmeyer, Karl Koscher, Stephen Checkoway, Stefan Savage, Aaron Schulman, Alex C Snoeren, and Kirill Levchenko. Triton: A software-reconfigurable federated avionics testbed. In *Proceedings of CSET 2019*, 2019.

[14] Frank da Cruz. Kermit protocol manual. `http://www.kermitproject.org/kproto.pdf`, June 1986.

[15] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of USENIX Security 2013*, pages 463–478, August 2013.

[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of TACAS 2008*, page 337–340, April 2008.

[17] *Devicetree Specification Release v0.2*. devicetree.org, December 2017. `https://github.com/devicetree-org/devicetree-specification/releases/download/v0.2/devicetree-specification-v0.2.pdf`.

[18] Bo Feng, Alejandro Mera, and Long Lu. P^2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of USENIX Security 2020*, August 2020. To appear.

[19] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8:156–166, March 1977.

[20] Fred Glover and Eric Taillard. A users guide to tabu search. *Annals of Operations Research*, 41:1–28, May 1993.

[21] Microcontroller Soluctions Group. *MCF54455 Reference Manual*. Freescale Semiconductor, March 2012.

[22] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of RAID 2019*, pages 135–150, September 2019.

[23] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin Butler. FirmUSB: Vetting USB device firmware using domain informed symbolic execution. In *Proceedings of CCS 2017*, October 2017.

[24] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 329–340, 2014.

[25] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *Proceedings of WOOT 2015*, August 2015.

[26] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of SAS 2011*, pages 95–111, September 2011.

[27] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[28] *BCM2835 Readme*. Raspberry Pi Foundation, February 2012. `https://web.archive.org/web/20200213201523/https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/README.md`.

[29] *BCM2836 Readme*. Raspberry Pi Foundation, August 2014. `https://web.archive.org/web/20200213201454/https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md`.

[30] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. SymDrive: Testing drivers without devices. In *Proceedings of OSDI 2012*, pages 276–292, October 2012.

[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of Security and Privacy 2016*, pages 138–157, May 2016.

[32] *OMAP35x Applications Processor: Technical Reference Manual*. Texas Instruments, December 2012.

[33] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. *Proceedings of CSUR 2020*, 54(1):1–36, 2021.

[34] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of NDSS 2014*, February 2014.

[35] Michal Zalewski. Technical "whitepaper" for afl-fuzz. `http://lcamtuf.coredump.cx/afl/technical_details.txt`, 2017.

[36] Michal Zalewski. The bug-o-rama trophy case. `http://lcamtuf.coredump.cx/afl/`, 2019.

A I/O Traces

Tables 3, 4 and 5 show inferred MMIO traces references in Sections 6.2.3, 6.4.3, and 6.5.3.

Table 3: The synthesized device traces for the Raspberry Pi 2. See text in Section 6.2.3 for discussion. Note that device register names and their function are included for exposition only. This information is *not* used by Jetset during inference or synthesis.

Device	Offset	I/O Behavior	Name *(# Reads)* Explanation
DMA Controller	0x220--0xE20	0x10000000	X_DEBUG *(10)* Reduced performance LITE engine
3F00 7000			enabled for DMA channels 0, 2, 5, and 8–14.
RNG	0x0	0x1	RNG_CTRL *(1)* RNG Unit Read Enabled
3F10 4000	0x4	0x10000000 0x0	RNG_STATUS *(2)* First read satisfies wait period for
			entropy, second specifies not to read from the RNG unit.
GPIO	0xC	0x0 0x8000 0x20000	GPFSEL3 *(3)* Initial control toggle, then GPIO pin 45 must
			be set as output, pin 45 to alternate function 0.
3F20 0000	0x10	0x1000000 0x20000000 0x1000000 0x0	GPFSEL4 *(8)* Various mode set requirements for GPIO
		0x0 0x800000 0x4000000 0x20000000	GPIO pins 40–49.
	0x14	0x4 0x20 0x100 0x800 0x0 0x0 0x0	GPFSEL5 *(12)* Similar to above, modes for GPIO pins 50–53
		0x0 0x4 0x20 0x100 0x800	(remaining bits not used)
	0x38	0x0 0x0 0x0	GPLEV1 *(3)* Pin levels all low or unconstrained reads.
	0x40	0x80000000	GPEDS0 *(1)* Event must be detected on GPIO pin 31!
	0x44	0xfd000000	GPEDS1 *(1)* "Reserved" event status bits high during
			clockevents_program_event .
UART0 (PL011)	0x0	0x0 ...	DR *(32)* No data recieved from UART peripheral.
3F20 1000	0x18	0x20 0x0 ... 0x20 0x0 ...	FR *(204)* Occasionally the UART transmit FIFO is full for
			a shorter path, otherwise flag register is empty.
	0x30	0x0 0x0 0x0 ...	CR *(1076)* Unconstrained read, modify, writes during serial
			printk statements.
	0x38	0x0	IMSC *(1)* UART interrupt mask 0 avoids longer print path.
SDHOST	0x0	0x0 0x0 0x0 0x8000 0x8000 0x0 0x0	SDCMD *(7)* Unconstrained reads during a command request,
			then a NEW_COMMAND read flag set, causing the request to fail.
3F20 2000	0x4--0x1C 0x30	0x0	SDARG, SDTOUT, SDCDIV ... *(11)* unconstrained reads during
	0x38 0x3c 0x50		fail-mode register dump.
	0x20	0x8 0x0	SDHSTS *(2)* FIFO error from SDHOST status, then
			unconstrained read during register dump.
	0x34	0x0 0x0 0x0 0x0	SDEDM *(4)* Unconstrained reads during enable, register dump.
AUX	0x4	0x4 0x0 0x2 0x0 0x1 0x0	AUX_ENABLES *(6)* Device reads SPI2, SPI1, and MINIUART
3F21 5000			during boot checks.

Table 4: The synthesized device traces for the CMU-900. See text in Section 6.4.3 for discussion. Note that device register names and their function are included for exposition only. This information is *not* used by Jetset during inference or synthesis.

Device	Offset	I/O Behavior	*(# Reads)* Explanation
Serial controller (Z85C30)	0x4	0x4 ... 0x4 0x0 0x4 0x0	*(215)* Read Register 0, 0x4 indicates the TX buffer is empty.
0x2000		0x0 0x4 ...	When needed, Jetset responds with a non-empty status.
Real-time clock (DS1685)	0x1	0x80 0x56 0x55 0x43 0x43	*(5)* Indicates that the power status is healthy, other values
0x3000			infer proper reads from CMOS user RAM.
I/O Processor (386ex)	0x0	0x0 0x0 0x0 0x0 0x0	*(5)* Unclear; first four act as a 4 I/O clock cycle delay,
0x5000			and the fifth errors if the fifth LSB is not zero.
Unknown	0x0	0x1 ... 0x1 0x0 0x0 0x20	*(16)* Reverse engineering finds a loop until a zero is read. The
0x6000		0x0	remaining values satisfy flags to skip additional configuration.
Discrete input status?	0x0	0x20	*(1)* The Airborne Data Loader (ADL) is disconnected.
0x21e40080			

Table 5: Some of the synthesized device traces for the SEL-751. See text in Section 6.5.3 for discussion. Note that device register names and their function are included for exposition only. This information is *not* used by Jetset during inference or synthesis.

Device	# Reads	Offset	I/O Behavior	Explanation
FPGA Related Space	343	0x0	0x0, 0x16e360, 0x0, 0x42fcf44e, 0x127fbe3c,	Interactions including synchronization of
0x30000000			0x19bf8e3c, 0x0, 0x1f4, 0x0, 0x1f3, 0x1f4, 0x0, 0x1f4, 0x0, 0x1f3, 0x1f4, 0x0, ...	the FPGA clock. Exact details unknown.
FlexBus Controller	215	0x4	0x0 ...	CSMR0 Unconstrainted reads during toggling
0xfc008000				of Flexbus device 0 read-only bit.
Fast Ethernet Controller 0	9	0x4	0x0, 0x0, 0x800000, 0x0, 0x0, 0x800000,	EIR0 Ethernet (MII) Interrupts are intermittently
0xfc030000			0x0, 0x0, 0x800000	raised, indicating a complete data transfer.
	3	0x40	0x610, 0x141, 0x1080	MMFR0 MII Frame Register reads correspond with raised interrupts, providing (invalid) data reads.
I^{2}C	9	0x8	0x0 ...	I2CR I2C control register—unconstrained reads
0xfc058000				occur while the firmware does control bit toggling.
	31	0xC	0x20, 0x0, 0x0, 0x0, 0x2, 0x0, 0x10, 0x0, ...	I2SR Status register interrupt and condition flags.
	4	0x10	0x0, 0x0, 0x0, 0x0 ...	I2DR Unconstrained data read from I/O Register.
GPIO Pin Mux and Control	6	0x32	0x0, 0x0, 0x0, 0x4, 0x0, 0x0	PPDSDR_SSI Serial pin reads for path satisfaction.
0xfc0a4000	1	0x34	0x0	PPDSDR_BE Bit read test expected to be 0.
	38	0x3c	0x10, 0x0, 0x10, 0x0, 0x10, 0x10, 0x10, ...	PPDSDR_PCI PCI data expected on pin 4.