SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems

Karl Koscher
UC San Diego
supersat@cs.ucsd.edu

Tadayoshi Kohno
University of Washington
yoshi@cs.washington.edu

David Molnar
Microsoft
dmolnar@microsoft.com

ABSTRACT

Embedded systems are becoming increasingly sophisticated, inter-connected, and pervasive. Unfortunately, securing these systems remains challenging. While powerful dynamic analysis tools have been developed for traditional software, the unique characteristics of embedded systems make it difficult to apply these well-known techniques; prior work has been limited either to small systems or short segments of code. In this paper, we demonstrate a system that is capable of emulating and instrumenting embedded systems in near-real-time, enabling a variety of dynamic analysis techniques. Our approach uses a custom, low-latency FPGA bridge between the host's PCI Express bus and the system under test, allowing the emulator full access to the system's peripherals. This provides the emulator with a faithful representation of the environment the firmware normally executes in, enabling additional dynamic analysis techniques such as concolic execution. We discuss the design decisions and engineering tradeoffs made and evaluate our system against prior work.

1. INTRODUCTION

Embedded systems are becoming increasingly sophisticated, inter-connected, and pervasive, making the "Internet of Things" the buzzword *du jour*. Unfortunately, these systems have repeatedly been shown to be insecure, with vulnerabilities in a diverse range of products such as automobiles [1], medical devices [2], routers [3], and voting machines [4]. Even if we can convince manufacturers to invest the time and resources to secure their products, the security tools available to embedded systems developers pale in comparison to those for traditional software.

In particular, dynamic analysis techniques are challenging to apply due to the difficulty of instrumenting embedded systems. There may not be sufficient storage space for an instrumented binary or its measurements. There may not be sufficient processing power for instrumentation. There may not be a way to provide arbitrary data to the system—a necessity for fuzzing. Even if a system is technically capable of added instrumentation, firmware heterogeneity requires substantial work to customize instrumentation for each device. Whereas traditional software runs on top of a few standard OSes (with standard facilities that support instrumentation, such as a file system and dynamic linker), embedded systems may not even *have* an OS. The analyst must identify instrumentation points and storage available for measurements, and surgically insert code into the firmware.

An alternative to placing instrumentation on the device itself is to run the system under emulation. However, this introduces its own set of challenges. Embedded systems are highly intertwined with their environment, through sensors, actuators, and other interfaces. Furthermore, the peripherals that control these interfaces can vary a great deal from one device to another. Faithfully emulating these peripherals requires a great deal of work building customized solutions.

An early approach to this problem came in the form of *in-circuit emulators,* which are drop-in, hardware replacements for microprocessors. These are typically microprocessor cores identical to those being "emulated," with extra debugging signals *bonded-out* and connected to external analyzers. These analyzers can be used to examine and control the operation of the microprocessors. However, as processor speeds have increased, and as microcontrollers have evolved into full Systems-on-Chip (SoCs), hardware in-circuit emulators have been replaced by special debugging facilities built in to most modern microcontrollers and SoCs. These facilities, while useful for development and debugging, often do not readily lend themselves to supporting advanced dynamic analysis techniques, such as taint tracking, fuzzing, or concolic execution.

Another approach, as described in section 2, is to treat peripherals as unconstrained symbolic inputs. However, this relies on the analysis using symbolic execution. Unconstrained inputs can lead to state explosion, rendering this technique unsuitable for all but the smallest embedded systems.

We take a different approach. Like Avatar [5] (also described in section 2), we run the device's firmware

under emulation, directing peripheral I/O to the actual device, giving the emulated firmware a realistic view of its environment. This leverages the fact that many devices rely on a relatively small set of embedded processors; SoC manufacturers typically license a well-known CPU core and add their own custom peripherals.

However, there are a number of challenges in making this approach work without being prohibitively slow. Avatar attempts to overcome these challenges by limiting the amount firmware executed under emulation. However, this raises a number of additional problems. The analyst must have sufficient insight into operation of the firmware to decide which parts are interesting enough to run under emulation. Emulated code still executes slowly, so this technique may not work with timing-sensitive devices (such as a medical device with a watchdog coprocessor.) Furthermore, it doesn't provide a feasible way to do whole-system analysis.

Instead of limiting the scope of emulated execution, we introduce a system called SURROGATES, which can emulate *entire* systems in *near-real-time*. We accomplish this by using custom, low-latency hardware to bridge the PCI Express bus of the host to the device under test, as well as making a number of optimizations. In doing so, we uncover and surmount new challenges in emulating entire systems, such as handling interrupts, DMA, and clocking changes.

In this paper, we make the following contributions: 1) We describe new hardware which enables near-real time emulation of arbitrary ARM-based embedded systems, providing a platform to build advanced dynamic analysis tools on; 2) We discuss the engineering tradeoffs in building SURROGATES and provide comprehensive performance evaluations of the different techniques; 3) We describe and solve several issues that arise when emulating entire systems; and 4) We demonstrate the practicality of using our system on a diverse set of devices.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 discusses a number of options to improve the performance of systems like Avatar, guiding the design of SURROGATES, which is introduced in section 4. Section 5 evaluates the performance of our system, compares it to prior work, and describes our experience applying our system to a variety of embedded systems. Section 6 describes future work. Finally, we conclude in section 7.

2. RELATED WORK

The poor state of embedded security and the seriousness of its consequences have led researchers to propose new ways to *automatically* analyze embedded systems, building on the success of traditional dynamic analysis tools. However, there are a number of challenges in applying traditional dynamic analysis tools to embedded systems.

Whereas traditional software is written against OS-provided APIs, the "API" that firmware is written against is usually a hardware specification. Peripherals typically expose their behavior through several *memory-mapped* registers. These registers appear as normal memory, but reads and writes to these addresses directly control the hardware. With the large heterogeneity of embedded devices, faithfully reproducing hardware behavior to dynamic analysis tools is a time-consuming and error-prone proposition.

FIE [6] symbolically executes the firmware of small, MSP430-based embedded devices. FIE overcomes the challenges in the diversity of devices and the need to understand peripheral semantics by treating *all* peripheral I/O as an unconstrained symbolic input. Unfortunately, this can easily lead to a state space explosion, making this technique impractical for all but the smallest embedded systems.

Avatar [5] attempts to constrain the number of states explored by using the *actual hardware* as a guide for peripheral semantics. It does so by redirecting peripheral I/O to the real device, either by using a JTAG debugger or through serial communication with an in-memory stub loaded onto the target in a manner similar to SerialICE [7]. Unfortunately, with the ability to do only about five memory operations per second, redirecting all I/O is prohibitively slow. Avatar overcomes this limitation by migrating executing code between the emulator and the device, and emulating only small portions of interest of the firmware. However, this optimization is unsuitable for timing-sensitive systems. We seek to overcome this limitation by enabling *near-real-time* peripheral interaction.

3. TOWARDS REAL-TIME I/O

Our system targets ARM processors, which are ubiquitous in medium-to-high complexity embedded devices. Our system communicates over the JTAG interface exposed on most microcontrollers. JTAG has several nice properties: 1) it is usually present in embedded devices for programming and testing during manufacturing, 2) JTAG pins are usually dedicated for programming and debugging, so it provides a communications channel that is not already used for

some other purpose during normal operation, 3) JTAG interfaces tend to support high transfer rates (e.g. ARM processors can support JTAG clock rates up to 1/6th of the core processor speed), limited primarily by off-chip factors such as connection length, and 4) existing JTAG tools can be used to read and write arbitrary memory addresses on a device, making it easy to rapidly develop an Avatar-like prototype.

JTAG interfaces expose a simple, standard state machine that can be driven by a JTAG adapter. This state machine lets the JTAG adapter select, capture, and update either a JTAG instruction register or a data register. These registers act like shift registers; data is shifted in and out simultaneously. While there is only one instruction register, several different data registers (called scan chains) can be selected using the different JTAG instructions.

As with Avatar, we first redirected emulated memory-mapped I/O to the target over JTAG using OpenOCD [8] (an open-source JTAG program). We initially used OpenOCD's built-in GDB protocol interface to initiate reads and writes and control the processor's state. However, memory operations are extremely slow over regular JTAG interfaces. This is because these memory operations are typically *injected* into the CPU's state. The JTAG interface must halt the CPU, transfer the CPU's state, update the CPU's state to perform a memory operation (including general purpose registers and the instruction register), single-step the CPU, transfer out the CPU's state again if the memory operation was a read, restore the CPU's original state, and resume the CPU.

While exposing the CPU's state over JTAG gives debuggers extremely powerful control over the system, its performance is poor for common tasks, such as transferring large segments of memory. To improve performance of these operations, CPU vendors have introduced additional scan chains that expose small communications channels between the JTAG interface and a program running on the CPU. For example, most ARM processors support the Debug Communications Channel (DCC), which is a 32-bit register accessible over a separate JTAG scan chain. JTAG interfaces can upload a small stub to the target and use the DCC to transfer large portions of memory efficiently.

We leverage the relatively fast DCC by developing a custom stub that runs on the target, accepting memory read and write commands from the host. A full discussion of our stub and DCC protocol is in section 4.2. We modified QEMU [9] to directly pass selected reads and writes as DCC commands to a Segger J-Link, a commercial, off-the-shelf USB JTAG interface. Unfortunately, we then encountered an unexpected bottleneck: USB transaction latency. USB requires all communications to be initiated by the host. This requires the host to periodically poll all devices for their status. The maximum polling rate is 1 kHz, which imposes a minimum latency of 1 ms on each USB transaction. While this may sound insignificant, it is several orders of magnitude slower than the latency of native I/O operations. Furthermore, because code execution may depend on the result of a memory read, this effectively places an upper-limit on the number of memory operations we can perform per second. Note that while we could continue to execute symbolically (later replacing the symbolic result of the read with its concrete value and pruning inconsistent code paths), further interactions with the hardware may depend on the result of the read, and thus to ensure consistency we must wait for the read to complete. This latency is a fundamental limitation of USB, which means that we must look at other interfaces to overcome it.

4. OUR APPROACH: SURROGATES

We decided to avoid further unexpected bottlenecks and latencies that might be lurking in other interfaces (such as Ethernet and Firewire) by developing a custom JTAG adapter that connects directly to the host's PCI Express bus. Our goal was to transparently map the target's entire 32-bit physical address space into the 64-bit address space of the emulator, such that peripheral I/O is simply a memory read or write by the emulator. While practical reasons (explained later in this section) prevent us from achieving this goal, our JTAG interface is directly memory-mapped into the emulator process, giving us extremely low-latency access to the target. We still use our DCC stub to communicate with the target processor.

The PCI Express bus is not really a bus at all, but a packet-switched network. The *root complex* translates CPU reads and writes into PCI Express packets, which get routed by address. (Alternate routing schemes can be used, e.g., for device discovery and configuration.) Writes are *posted* transactions which complete immediately, while reads are *unposted*, which require a completion packet (usually with data) to be sent back to the root complex. Since PCI Express is a packet-switched network, devices can send packets to their peers, as well as performing DMA by sending packets to the root complex.

4.1 The Hardware

Our hardware consists of an off-the-shelf PCI Express FPGA card (a Pico Computing E17FX70T), a custom FPGA-to-JTAG interface board, and a custom JTAG debugging board, as shown in Figure 1. The FPGA-to-

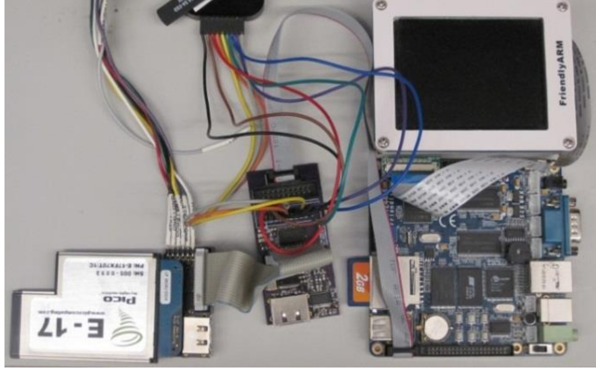

Figure 1: Hardware components of our system. Left-to-right: An off-the-shelf FPGA ExpressCard, our JTAG adapter board, a JTAG breakout/debug board, and the device under test (a FriendlyARM Mini2440). FPGA development and debugging is done through another JTAG connection via the JTAG interface board, as well as a small logic analyzer connected to the JTAG breakout/debug board.

JTAG board shifts signal voltage levels between the FPGA and the target's JTAG interface, and provides a standard ARM JTAG connector. It also provides a SATA-like, high-speed serial interface that can transport JTAG signals over a longer distance. The JTAG debugging board can convert this serial stream back to a standard JTAG interface, and provides an easy interface for a logic analyzer to examine the JTAG signals.

Our implementation uses a Xilinx Virtex5 FX70T FPGA. While this FPGA is overkill for our purposes, it was available off-the-shelf as a PCI Express card, with the bulk of the PCI Express glue logic already developed by Xilinx and Pico Computing. Our application logic is implemented in approximately 1,100 lines of Verilog, excluding tests (which are approximately another 1,000 lines of Verilog). Device utilization is summarized in Table 1.

We implement two PCIe-to-JTAG bridges in the FPGA. The first is a simple set of FIFOs for the TDI, TMS, and TDO signals, and supports generic JTAG operations, such as manipulating the processor's state, dumping firmware, and uploading code. We extend OpenOCD to support this new interface and use it for some complicated-but-infrequent operations, such as resetting the target to a known state and uploading the stub.

The second interface is designed specifically to work with our stub. As previously mentioned, the original intention was to provide a transparent mapping of the

Table 1: FPGA Utilization

	Used	*Available*	*Utilization*
Slice Registers	6,503	44,800	14%
Slice LUTs	6,615	44,800	14%
Occupied Slices	3,397	11,200	30%
BlockRAMs/FIFOs	11	148	7%
Total Memory (KB)	306	5,328	5%

target's 32-bit physical address space somewhere in the host's 64-bit address space. Unfortunately, the PCI Express specification requires that all 64-bit address ranges be *prefetchable*—meaning that reads are side-effect free. This is not the case for several embedded devices. For example, a UART controller may have a single, memory-mapped character register. A read from this register frees the UART to receive another byte. While some chipsets do allow 64-bit PCI Express regions to not be prefetchable, others do not.

Of course, only a portion of the target's 32-bit address space is mapped to peripherals. We considered transparently mapping a small view of the target's address space, allowing the host to pick the address range that is mapped in. However, on a typical PC, there is a great deal of contention for address space below the 4GB boundary. This makes it difficult to map reasonably large 32-bit regions. Furthermore, devices typically use large peripheral address spaces (e.g. 320 MB on the Samsung S3C2440) even though they are sparsely populated. Since the host may have to keep remapping different views of the target's address space, we decided to simply expose a few memory-mapped registers that initiate reads and writes to the target. These registers are described below and shown in Appendix A.

There are two address registers–one for reads, and one for writes, as well as a data register. When a write address and value are written, the FPGA initiates a write operation on the target through its DCC interface. When an address is written to the read address register, a read operation on the target is initiated. We also provide two FIFOs and control registers to allow the host to initiate optimized multiple-word transactions.

The packet-based nature of PCI Express lets us stall reads of the data register if the target hasn't returned data yet. However, while the root complex is supposed to abort transactions that have timed out, our particular root complex doesn't. This means that if the target

device doesn't respond (due to a bug, being powered off, etc.), the host will freeze. Not even the NMI watchdog can recover the system. For this reason, we typically poll the FPGA for completion.

When there are no pending read or write requests, the FPGA can be configured to continuously poll the target's DCC register to see if an interrupt has occurred. Interrupts received from the stub are dispatched as interrupts to the host's processor. This required a small modification to the FPGA's PCI Express interface code. The preferred way of sending interrupts over PCI Express is to use *Message Signaled Interrupts (MSIs),* which are simply memory writes of a specific value to a specific address. Peripherals no longer have to share a total of four interrupt signals, and can in fact request multiple interrupts. This would appear to allow the hardware to send different interrupts to the host based on the target's interrupt type. Unfortunately, Linux has limited support for multiple interrupts per peripheral, so the driver must poll the hardware to determine the interrupt type, as described in section 4.3.

4.2 The Stub

Our stub targets most microcontrollers based on ARMv4T or newer cores. (Some newer ARM Cortex cores have different debugging options and capabilities.) This covers a wide range of interesting embedded devices, including hard drives, cellular baseband processors, medical devices, and automotive systems. The stub is implemented in approximately 400 lines of assembly and takes up only 768 bytes—which can be easily locked into the instruction cache on processors that support it. The stub does not use any RAM for data or a stack, allowing the emulator to use all available RAM on the target if desired.

Our stub uses a custom word-based protocol to efficiently perform memory operations as well as transferring status information, such as interrupts and interrupt masks. A summary of our protocol is listed in Table 2.

The stub provides handlers for standard (IRQ) and fast (FIQ) interrupts. Unlike Avatar, no de-multiplexing is attempted. When an interrupt is received, ARM processors update their Current Program Status Register (CPSR) to set the IRQ or FIQ Disable bit, preventing the handler from being interrupted itself. The old CPSR value is stored in the Saved Program Status Register (SPSR). Normally when the handler returns, the SPSR is copied back to the CPSR, re-enabling interrupts. However, we adjust the SPSR to keep interrupts disabled and deliver the interrupt type

Table 2: Our stub protocol as 32-bit hex words

Words	Description
►`1SXXXXXX` ►`YYYYYYYY` ◄`ZZZZZZZZ` …	Read XX words of size S (1, 2, or 4 bytes) from address YY. XX data elements ZZ are returned.
►`2S00XXXX` ►`YYYYYYYY`	Write a single word XX of size S (1 or 2 bytes) to address YY.
►`3SXXXXXX` ►`YYYYYYYY` ►`ZZZZZZZZ` …	Write XX words of size S (1, 2, or 4 bytes) to address YY. XX data elements ZZ are sent.
►`50XXXXXX`	Set the CPSR register to XX. Primarily used to set and clear interrupt flags.
… ◄`C347A5XX` …	An interrupt of type XX has occurred. This word can be sent at any time, including before a read response. In the unlikely case that a word C347A5XX is the result of a read operation, C347A500 is sent as an escape sequence.

to the host. The host delivers the interrupt to the emulated processor when its CPSR is set to allow interrupts. The emulated firmware can then query the interrupt controller like any other peripheral to determine the source(s) of the interrupt. Note that multiple interrupt sources may be set in the interrupt controller–setting the IRQ or FIQ Disable flag does *not* mask interrupts from being handled by the interrupt controller, but merely prevents them from being delivered to the CPU. The firmware acknowledges any interrupts it handles. When the emulated firmware finally re-enables interrupts, a CPSR update command is sent to the target to re-enable its interrupts. If the interrupt controller still has an unacknowledged interrupt active, it will once again interrupt the target CPU. This process repeats until no interrupts are active. The acknowledgement protocol prevents any race conditions where the emulated processor may miss an interrupt. Since these race conditions can appear natively, all ARM firmware must implement this type of protocol. Some ARM SoCs provide vectored interrupts, where the firmware can specify different handlers for each interrupt source. However, since the ARM core itself only supports two interrupt types, these vectors are normally implemented with a small handler in ROM, which queries the interrupt controller and jumps to the correct vector. This ROM can be emulated by our system like any other

firmware, allowing us to support fully-vectored interrupts with no additional work. Extracting this ROM and other per-device setup is discussed in section 5.2.

4.3 The Software

We modified QEMU [9] to pass all MMIO to our hardware. We accomplished this by creating a new "surrogate" peripheral in QEMU, which owns the entire MMIO address space of the target and forwards MMIO operations to the hardware. We also created a new QEMU "system," which selects the proper CPU, creates the necessary address spaces, initializes the surrogate peripheral, and loads the firmware to emulate. Note that since we build on QEMU, our system easily integrates with tools such as S^2E [10] and Avatar. (We later created interfaces to our hardware as S^2E and Avatar plugins, but found that doing so incurs a substantial performance hit. Thus, we appear to S^2E like any other virtualized peripheral.)

Initially we ran our system under Windows to take advantage of the existing drivers for the PCIe card. However, the drivers were optimized for streams of data, where latency is less of a concern that throughput. For example, transfers to the card would always use DMA, regardless of the transfer size.

We ultimately re-implemented a simplified version of the driver on Linux (which was based on an open-source driver for Pico Computing's other FPGA products). To avoid syscall overhead on every MMIO operation, we allow applications to mmap the hardware's register space, although in practice this did not significantly improve performance.

Finally, we extended the driver's interrupt handler to deliver a signal to any process that requests it whenever a non-DMA interrupt is received. A signal handler in QEMU delivers this interrupt to the virtual CPU. This provides a low-latency path for interrupts.

5. Evaluation

We evaluate our system against two metrics: its performance and the ease of configuring it to work with a new target device.

5.1 Performance

One of the key motivations for SURROGATES was to overcome the performance limitations of Avatar. While we had independently built a system very similar to Avatar, we were unable to use it against several devices of interest because proper operation of those devices relies on timing constraints that it could not meet (e.g. watchdogs on co-processors of a medical device). Therefore, we evaluate the several

Table 3: Raw MMIO Performance

	MMIO Operations Per Second
Avatar	~5 (over serial debug port at 38400 bps)
Our system w/ syscalls	17172 writes / 15761 reads (over 4 MHz JTAG)
Our system w/ mmap	17174 writes / 15772 reads (over 4 MHz JTAG)

performance aspects of our system and compare it with prior work. All of our performance experiments were run against a FriendlyARM Mini2440 development board, described in Section 5.2.

To test raw MMIO performance, we measure the time needed to make 1,000,000 read or write requests to the SRAM of the FriendlyARM's SoC, connected to our hardware with a 4 MHz JTAG clock. We find that our raw MMIO performance is four orders of magnitude faster than what the Avatar authors reported, as shown in Table 3. We also measured the time taken to write to an FPGA register 1,000,000 times. Although accessing the FPGA through a mmap interface is about 60% faster (1.4 µs vs. 2.2 µs), the overall performance impact under real workloads is negligible.

To evaluate whether this performance was reasonable to support near-real-time emulation, we set out to boot Linux on the emulated processor. To accurately measure the amount of time to boot, we replaced the init binary with one that simply contains a special illegal instruction. This instruction shuts down QEMU and reports performance statistics. We found that the kernel boots in about 27 seconds. 25 seconds were spent performing I/O. However, during boot the kernel initializes all of the peripherals, so its I/O characteristics are different from typical usage of a booted system. During this time, approximately 126,000 reads and 87,000 writes were performed.

To evaluate interactivity, we replaced the init binary with the busybox [11] version of /bin/sh, allowing us to interact with the system over its serial port. While file system accesses were noticeably slower than on the real hardware, the shell maintained a subjectively good amount of responsiveness.

To get a more objective measure of responsiveness, we connected the FriendlyARM's Ethernet port directly to a Windows laptop and performed a ping test against the emulated system. After 100 pings, the average response time was 15 ms. The minimum response time was 8 ms, and the maximum was 61 ms. We then connected the FriendlyARM to our campus network

(which has significantly more broadcast traffic) and obtained similar results. Finally, we loaded a web page from the emulated device's HTTP server, which loads content off of the physical SD card and sends it over the physical NIC. When loading a 369KB image from the SD card, we obtained an effective throughput of 17.3 KB/s, which includes an initial stall to read the file from the SD card. Subsequent transfers of the same image (now in the filesystem cache) had a throughput of about 26 KB/s. Note that neither the SD card driver nor the NIC driver use DMA, which would allow us to exploit the multi-word transfer mode of our system to approximately double our throughput (since we transfer the address only once, and not on every word transfer).

While slower than running natively, we are able to emulate an *entire* system with reasonable usability. In contrast, the authors of Avatar reported that it took almost four minutes to reach the bootloader prompt of a hard drive.

5.2 Portability

This work was also motivated by our desire to build a dynamic analysis platform that does not require a great deal of work to apply to a new target. Therefore, we evaluate the ease of supporting new devices and discuss some of the new challenges encountered when supporting entire systems. We look at two devices as case studies: a FriendlyARM Mini2440 development board with a Samsung S3C2440 SoC, and a wireless medical device with an iMX21 SoC.

When applying our system to a new target, the first task is to identify the target's JTAG port. These are often connected to test pads on the target's PCB, but sometimes they are brought out to dedicated connectors. As a development board, the FriendlyARM features a well-identified JTAG port. The wireless medical device, however, just has dozens of unmarked test points. We had previously identified the JTAG test points through manual analysis; however, today there are tools like the JTAGulator [12] that perform a brute-force search over all test points to find the JTAG signals.

Once JTAG connectivity is established, firmware of the device is downloaded. In some cases, the SoC itself has a small amount of firmware in ROM that is essential to proper operation of the SoC. For example, the ROM in the iMX21 performs interrupt vectoring, so if the firmware chooses to use vectored interrupts, the ROM must be emulated as well.

A location for the stub must be identified. Different SoCs have varying requirements for locating interrupt and exception handlers. For example, on the S3C2440, exception handlers must be located at 0x00000000, while on the iMX21, we can place exception handlers anywhere in memory because the ROM at 0x00000000 uses an exception vector table stored in dedicated RAM as a level of indirection. On the S3C2440, we place our stub in the NAND "SteppingStone" SRAM at 0x00000000. On the iMX21, we place our stub in the dedicated exception handler SRAM. Depending on the SoC, it may also be possible to lock the stub into the cache, allowing you to virtually place it over address spaces that are normally not usable (such as ROMs at 0). MMUs, if available, may also be used to place the stub at arbitrary locations, but this is left for future work.

Next, the layout of the target's address space must be specified in QEMU. Usually this is as simple as defining the address regions of RAM, Flash, and peripherals. For the iMX21, an additional address space entry is created for the ROM.

There are usually a few exceptions that must be carved out of the peripheral address space. These are for registers that, when updated, cause the target to lose sync with the host. For example, on the S3C2440, there are registers that control the core clock speed. When the clock speed is adjusted, the CPU is halted until the PLLs re-lock. JTAG communication fails until the CPU resumes execution. We can use dynamic analyses techniques to easily determine these exceptions. If we log all MMIO as the system boots, the last MMIO operation before the system halts is usually responsible for the failure. The SoC datasheet can be consulted for the effect of the corresponding register so that an intelligent exception can be made.

Finally, different SoCs have wildly varying DMA controllers, some of which must be emulated for proper emulation of the device. For example, the S3C2440 has a general-purpose DMA controller as well as a dedicated LCD DMA controller. Neither are *required* to be emulated to boot Linux. For the iMX21, we emulated the LCD DMA controller registers in QEMU with only eight additional lines of C. This emulated DMA controller simply copies the specified video memory from the emulator to the same location on the target, and then passes the DMA request on to the real DMA controller to transfer the data to the LCD.

As an alternative to emulating different DMA controllers, we can treat the emulator's memory as another level of cache. DMA controllers typically cannot access the L1 or L2 caches, so any data involved in a transfer must reside in main memory. We

can treat intentional cache invalidations as an indication that the memory was or will be used in a DMA transfer and flush the affected memory to or from the target. (Note that the stub always runs with the target's data caches off, so flushes from the emulator to the target will go directly to main memory). Unfortunately, this approach only works with firmware that turns the data caches on, which was not the case with our wireless medical device.

Overall, we find it straightforward to apply our system to different devices, requiring far less work than building an emulator for all of the target's hardware. There is some manual configuration involved, but this is true of most dynamic analysis tools.

6. Future Work

6.1 Further improving performance

While our stub protocol is relatively efficient, it still suffers from inefficiencies in ARM's DCC specification and limitations of JTAG interfaces. For example, to read a debug register, we must clock in 36 bits into the EmbeddedICE interface to select the register to read, and then clock another 36 bits out to read the value. There are two EmbeddedICE registers we use: the DCC status register, and the DCC data register. To read a single 32-bit value from the DCC data register, at least 144 bits need to be transferred. While we could propose some changes to the DCC specification, the most recent ARM processors have transitioned to debugging interfaces that provide complete access to the SoC bus. We have not yet examined these new interfaces in detail, as many systems of interest do not use them yet, but it may be straight forward to adapt our system to ARM's new debugging interfaces.

6.2 Eliminating our dependence on hardware

While our system enables dynamic analysis of embedded systems at an unprecedented scale, it doesn't necessarily scale any further. Systems like SAGE [13] and S^2E depend on the ability to massively parallelize state space searches. This is easy with well-defined OS APIs, but our approach depends on an individual physical system to guide execution. Even worse, to ensure the hardware is in a consistent state, we may need to reset the SoC and replay all I/O operations when another code branch is explored. (In practice, peripherals usually have limited state, so once they are initialized, we may be able to relax our consistency requirements and ignore their states.)

However, it may be possible to learn models of the hardware based on execution traces collected with our system. This would enable dynamic analysis systems to run largely independent of physical hardware, allowing it to scale up massively. The models do not necessarily need to be 100% accurate; as long as they reasonably constrain the state space search, it is feasible to explore several potentially vulnerable code paths. When a potentially vulnerable code path is found, it can be verified against the actual hardware using our system.

7. Conclusions

We have built and evaluated a system that enables dynamic analysis of embedded systems at an unprecedented scale. Our approach is similar to Avatar; we run the system under emulation in QEMU and redirect I/O to the target hardware to guide execution and provide the firmware with a faithful reproduction of its environment. However, by using a custom FPGA bridge between the host and target, we enable near-real time emulation of the target system, allowing us to analyze systems of far greater complexity. This will ultimately enable embedded systems developers to take advantage of several dynamic analysis techniques that were previously available only to traditional software developers, allowing them to deliver safer and more secure embedded systems.

8. References

[1] Stephen Checkoway et al., "Comprehensive Experimental Analyses of Automotive Attack Surfaces," in *USENIX Security Symposium*, San Francisco, 2011.

[2] Daniel Halperin et al., "Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses," in *IEEE Symposium on Security and Privacy*, 2008.

[3] Michael Lynn, "Cisco IOS Shellcode," in *Blackhat USA*, Las Vegas, 2005.

[4] Ariel J Feldman, Alex Halderman, and Edward W Felten, "Security Analysis of the Diebold AccuVote-TS Voting Machine," in *Electronic Voting Technology Workshop*, 2007.

[5] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti, "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Network and Distributed System Security Symposium*, 2014.

[6] Drew Davidson, Benjamin Moench, Somesh Jha, and Thomas Ristenpart, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *USENIX Security Symposium*, 2013.

[7] coresystems GmbH, SerialICE, 2009, http://ww.serialice.com/.

[8] Dominic Rath, Open On-Chip Debugger: Design and Implementation of an On-Chip Debug Solution for Embedded Target Systems, 2005.

[9] F. Bellard, et. al. QEMU. http://www.qemu.org/

[10] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea, "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems," in *6th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, 2011.

[11] BusyBox, http://www.busybox.net/.

[12] Joe Grand, JTAGulator, 2013, http://www.grandideastudio.com/portfolio/jtagulator/.

[13] Patrice Godefroid, Michael Y. Levin, and David Molnar, "Automated Whitebox Fuzz Testing," in *The 15th Annual Network & Distributed System Security Conference*, San Diego, 2008.

Appendix A: FPGA Register Map

<table>
<tr><th>Addr</th><th>Desc.</th><th>Value Specification</th></tr>
<tr><td>000</td><td>Output Control Register</td><td>Bits:
<table>
<tr><td>31-11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>Reserved</td><td>FORCE OUT</td><td>OUT EN</td><td>DBGACK</td><td>DBGRQ</td><td>nSRST</td><td>TDO</td><td>RTCK</td><td>TCK</td><td>TMS</td><td>TDI</td><td>nTRST</td></tr>
</table>
FORCEOUT – Forces JTAG output pins to the values set in this register
OUTEN – Enables JTAG output pins</td></tr>
<tr><td>004</td><td>JTAG Stream Control Register</td><td>Bits:
<table>
<tr><td>31-27</td><td>26</td><td>25</td><td>24</td><td>23-0</td></tr>
<tr><td>Reserved</td><td>Stub Interface Reset</td><td>Stub Interface Scan Enable</td><td>Stream Enable</td><td>Stream Length</td></tr>
</table>
Stub Interface Reset – Reinitializes the stub interface logic
Stub Interface Scan Enable – Causes the stub interface logic to poll the target for interrupts
Stream Enable – Streams arbitrary JTAG data (used for non-stub communication)
Stream Length – The number of bits to stream</td></tr>
<tr><td>008</td><td>JTAG Clock Divisor</td><td>Bits:
<table>
<tr><td>31</td><td>30-0</td></tr>
<tr><td>JTAG Clock Reset</td><td>Divisor</td></tr>
</table>
Divisor – The JTAG clock divisor. The JTAG clock speed is 125 MHz / (divisor – 1).</td></tr>
<tr><td>00C</td><td>Read Stall Control</td><td>Bits:
<table>
<tr><td>31</td><td>30-0</td></tr>
<tr><td>Read Stall Enable</td><td>Read Timeout</td></tr>
</table>
Read Stall Enable – Stalls reads from the Data Register until data is ready
Read Timeout – Read stall timeout, in multiples of 8 ns</td></tr>
<tr><td>x10</td><td>Read Address</td><td>Target address to read. X is the transfer size: 1 = Byte, 2 = 16 bit word, 4 = 32 bit word. Writes to this register initiate a read from the target.</td></tr>
<tr><td>x14</td><td>Write Address</td><td>Target address to write. X is the transfer size: 1 = Byte, 2 = 16 bit word, 4 = 32 bit word.</td></tr>
<tr><td>018</td><td>Data Register</td><td>Data returned from a read, or data to be written. Ignored in bulk transfer mode. Writes to this register always initiate a write to the target.</td></tr>
<tr><td>01C</td><td>IRQ Register</td><td>Bits:
<table>
<tr><td>31-8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3-0</td></tr>
<tr><td>Reserved</td><td>FIQ</td><td>IRQ</td><td>Reserved</td><td>Data Abort</td><td>Reserved</td></tr>
</table>
Reads from this register are unacknowledged exceptions received from the stub. Write a 1 back to the corresponding bit to acknowledge the exception.</td></tr>
<tr><td>024</td><td>Target CPSR</td><td>Writes to this register update the target's CPSR to the given value.</td></tr>
<tr><td>028</td><td>Bulk Data Length</td><td>Bits:
<table>
<tr><td>31-25</td><td>24</td><td>23-0</td></tr>
<tr><td>Reserved</td><td>BULKEN</td><td>Number of elements (bytes, half-words, words) to send</td></tr>
</table>
BULKEN – If set, the stub interface logic uses the bulk-optimized stub protocol, using the stub data FIFOs instead of the Data Register</td></tr>
</table>