Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares

Jonas Zaddach, Luca Bruno, Aurélien Francillon, Davide Balzarotti
EURECOM, France
{zaddach,bruno,francillon,balzarotti}@eurecom.fr

***Abstract*—To address the growing concerns about the security of embedded systems, it is important to perform accurate analysis of firmware binaries, even when the source code or the hardware documentation are not available. However, research in this field is hindered by the lack of dedicated tools. For example, dynamic analysis is one of the main foundations of security analysis, e.g., through dynamic taint tracing or symbolic execution. Unlike static analysis, dynamic analysis relies on the ability to execute software in a controlled environment, often an instrumented emulator. However, emulating firmwares of embedded devices requires accurate models of all hardware components used by the system under analysis. Unfortunately, the lack of documentation and the large variety of hardware on the market make this approach infeasible in practice.**

In this paper we present Avatar, a framework that enables complex dynamic analysis of embedded devices by orchestrating the execution of an emulator together with the real hardware. We first introduce the basic mechanism to forward I/O accesses from the emulator to the embedded device, and then describe several techniques to improve the system's performance by dynamically optimizing the distribution of code and data between the two environments. Finally, we evaluate our tool by applying it to three different security scenarios, including reverse engineering, vulnerability discovery and hardcoded backdoor detection. To show the flexibility of Avatar, we perform this analysis on three completely different devices: a GSM feature phone, a hard disk bootloader, and a wireless sensor node.

I. Introduction

An embedded system consists of a number of interdependent hardware and software components, often designed to interact with a specific environment (e.g., a car, a peacemaker, a television, or an industrial control system). Those components are often based on basic blocks, such as CPUs and bus controllers, which are integrated into a complete custom system. When produced in large quantities, such customization results in a considerable cost reduction. For large quantities, custom built integrated circuits (ASIC) are preferred as they allow to tailor functionality according to the specific needs, which results in cost reduction, better integration, and a reduction of the total number of parts. Such chips, also called System on a Chip (SoC), are often built from a standard CPU core to which both standard and custom hardware blocks are added. Standard blocks, commonly called *IP Cores*, are often in the form of a single component that can be integrated into a more complex design (e.g., memory controllers or standard peripherals). On the other hand, custom hardware blocks are often developed for a specific purpose, device, and manufacturer. For example, a mobile phone modem may contain a custom voice processing DSP, an accelerator for the GSM proprietary hardware cryptography (A5 algorithms) and an off-the-shelf USB controller.

Over the years, such SoCs have significantly grown in complexity. Nowadays, they often include Multiple Processors (MPSoC) and complex, custom, hardware devices. As a consequence, virtually every embedded system relies on a different, application specific, system configuration. As a witness of this phenomenon, the website of ARM Ltd., which provides one of the most common CPU core used in embedded systems, lists about 200 silicon partners[1]. Most of those partners are producing several product families of SoCs relying on ARM cores. This leads to a huge number of systems on the market, which are all different, but all rely on the same CPU core family.

Unfortunately, the increasing pervasiveness and connectivity of embedded devices significantly increased their exposure to attacks and misuses. Such systems are often designed without security in mind. Moreover visible features, low time to market, and reduction of costs are the common driving forces of their engineering teams. As a consequence, an increase in the number of reports of embedded systems exploitation has been recently observed, often with very serious consequences [8], [11], [12], [19], [23], [25], [44], [46], [54], [60]. To make things worse, such systems frequently play an important role in security-relevant scenarios: they are often part of safety critical systems, integrated in home networks, or they are responsible to handle personal user information. Therefore, it is very important to develop the tools and techniques that would make easier to analyze the security of embedded systems.

In the traditional IT world, dynamic analysis systems play a crucial role in many security activities - ranging from malware analysis and reverse engineering, to vulnerability discovery and incident handling. Unfortunately, there is not an equivalent in the embedded system world. If an attacker compromises the firmware of a device (e.g., a smart meter or a PLC in a Stuxnet-like attack scenario [25]) even vendors often do not have the required tools to dynamically analyze the behavior of the malicious code.

Dynamic analysis allows users to overcome many limitations of static analysis (e.g., packed or obfuscated code) and to perform a wide range of more sophisticated examinations [24] - including taint propagation [33], [55], symbolic and concolic execution [10], [15], [22], unpacking [34], malware sandboxing [1], [5], and whitebox fuzzing [28], [29].

Unfortunately, all these techniques and their benefits are still not available in the world of embedded systems. The reason is that in the majority of the cases they require an

[1] http://www.arm.com/community/partners/silicon.php

emulator to execute the code and possibly monitor or alter its execution. However, as we will explain in Section II, the large number of custom and proprietary hardware components make the task of building an accurate emulator a daunting process. If we then consider that additional modules and hardware plugins should be developed for each embedded system on the market, we can easily understand the infeasibility of this approach.

In this paper, we present a technique to fill this gap and overcome the limitation of pure firmware emulation. Our tool, named *Avatar*, acts as an orchestration engine between the physical device and an external emulator. By injecting a special software proxy in the embedded device, *Avatar* can execute the firmware instructions inside the emulator while channeling the I/O operations to the physical hardware. Since it is infeasible to perfectly emulate an entire embedded system and it is currently impossible to perform advanced dynamic analysis by running code on the device itself, *Avatar* takes a hybrid approach. It leverages the real hardware to handle I/O operations, but extracts the firmware code from the embedded device and *emulates* it on an external machine.

To summarize, in this paper we make the following contributions:

- We present the design and implementation of *Avatar*, a novel dynamic analysis framework that allows a user to emulate the firmware of an embedded device.
- We discuss several techniques that can be used to optimize the performance of the system and to adapt *Avatar* to the user's needs. We also show how complex dynamic analysis applications (such as concolic execution) can be implemented on top of *Avatar*.
- We evaluate *Avatar* by applying it to three different security scenarios, including reverse engineering, vulnerability discovery, and backdoor detection. To show the flexibility of our system, each test was performed on a completely different class of devices.

II. Dynamic Firmware Analysis

While the security analysis of firmwares of embedded devices is still a new and emerging field, several techniques have been proposed in the past to support the debugging and troubleshooting of embedded systems.

Hardware debugging features (mostly built around In-Circuit Emulators [13], [35], [42] and JTAG-based hardware debuggers [3]) are nowadays included in many embedded devices to simplify the debugging procedure. However, the analysis remains extremely challenging and often requires dedicated hardware and a profound knowledge of the system under test. Several debugging interfaces exist, like the Background Debug Mode (BDM) [58] and the ARM CoreSight debug and trace technology [58]. Architecture-independent standards for debugging embedded devices also exist, such as the IEEE NEXUS standard [4]. Most of these technologies allow the user to access, copy, and manipulate the state of the memory and of the CPU core, to insert breakpoints, to single step through the code, and to collect instructions or data traces.

When available, hardware debugging interfaces can be used to perform certain types of dynamic analysis. However, they are often limited in their functionalities and do not allow the user to perform complex operations, such as taint propagation or symbolic execution. In fact, these advanced dynamic analysis techniques require an instruction set simulator to interpret the firmware of the embedded target. But for a proper emulation of the embedded system, not only the CPU, but all peripheral devices need to be emulated. Without such a support, the emulated firmware would often hang, crash, or in the best case, show a different behavior than on the real hardware. Such deviations can be due, for example, to incorrect memory mappings, active polling on a value that should be changed by the hardware, or the lack of the proper hardware-generated interrupts or DMA operations.

To overcome these problems, researchers and engineers have resolved to three classes of solutions, each with its own limitations and drawbacks:

- *Complete Hardware Emulation*
  Chipounov [14] and Kuznetsov et al. [37] analyze device drivers by relying on an emulated PCI bus and network card that return symbolic values. This approach has the main drawback that it requires to emulate the device properly. While this is not much of a problem for well understood devices, like a PCI network card supported by most PC emulation software, it can be a real challenge in embedded systems and can be just impossible when the hardware is not documented. Unfortunately, lack of documentation is the rule in the embedded world, especially in complex proprietary SoCs.
  In some cases, accurate system emulators are developed as part of the product development to allow the firmware development team to develop software while the final hardware is still not available. However, those emulators are usually unavailable outside the development team and they are often not designed for code instrumentation, making them unable to perform basic security analysis like tainting or symbolic execution.

- *Hardware Over-Approximation*
  Another approach consists in using a generic, approximated, model of the hardware. For example, by assuming interrupts can happen at any time or that reading an IO port can return any value. This approach is easy to implement because it does not require a deep knowledge of the real hardware, but it can clearly lead to false positives, (e.g., values that will never be returned by the real system) or misbehavior of the emulated code (when a particular value is required). This approach is commonly used when analyzing small systems and programs that are typically limited to a few hundreds lines of code, as showed by Schlich [49] and Davidson et al. [22]. However, on larger programs and on complex peripherals this approach will invariably lead to a state explosion that will prevent any useful analysis.

- *Firmware Adaptation*
  Another approach consists in adapting the firmware (or in extracting limited parts of its code) in order to emulate it in a generic emulator. While this is possible in some specific cases, for example with Linux-based embedded devices, this technique does not allow for an holistic analysis and may still be limited by the presence of custom peripherals. Moreover, this approach is not possible for monolithic firmwares that cannot be easily split into independent parts - unfortunately a very common case in low-end embedded systems [20].

In the next section we present our novel hybrid technique based on a combination of the actual hardware with a generic CPU emulator. Our approach allows to perform advanced dynamic analysis of embedded systems, even when very little information is available on their firmware and hardware, or when basic hardware debugging support is not available. This opens the possibility to analyze a large corpus of devices on which dynamic analysis was not possible before.

III. AVATAR

Avatar[2] is an event-based arbitration framework that orchestrates the communication between an emulator and a target physical device.

Avatar's goal is to enable complex dynamic analysis of embedded firmware in order to assist in a wide range of security-related activities including (but not limited to) reverse engineering, malware analysis, vulnerability discovery, vulnerability assessment, backtrace acquisition and root-cause analysis of known test cases.

A. System Architecture

The architecture of the system is summarized in Figure 1: the firmware code is executed inside a modified emulator, running on a traditional personal computer. Any IO access is then intercepted and forwarded to the physical device, while signals and interrupts are collected on the device and injected into the emulator.

The internal architecture is completely event-based, allowing user-defined plugins to tap into the data stream and even modify the data as it flows between the emulator and the target.

In the simplest case *Avatar* requires only a backend to talk to the emulator and one to talk to the target system, but more plugins can be added to automate, customize, and enhance the firmware analysis. In our prototype, we developed a single emulator backend. This controls S^{2}E (or Selective Symbolic Execution engine), which is an open-source platform for selective symbolic execution of binary code [15]. It builds on the foundation of Qemu, a very popular open-source system emulator [7]. Qemu supports many processor families such as i386, x86-64, Arm, Mips and many others. Apart from being a processor emulator, Qemu can also mimic the behavior of many hardware devices that are typically attached to the central processor, such as serial ports, network cards, displays, etc.

S^{2}E leverages the intermediate binary code representation of Qemu called Tiny Code Generator (TCG), and dynamically translates from TCG bytecode to Low-Level Virtual Machine (LLVM) bytecode whenever symbolic execution is active [39]. KLEE, the actual symbolic execution engine, is then taking care of exploring the different execution paths and keeps track of the path constraints for each symbolic value [10]. Evaluating possible states exhaustively, for some symbolic input, can be assimilated to model checking and can lead to proving some property about a piece of software [38].

Even though S^{2}E uses the TCG representation of the binary code to generate LLVM code, each processor architecture has its own intricacies that make it necessary to write architecture specific extensions to make S^{2}E work with a new processor architecture. Since our focus was on embedded systems and all the systems we analyzed are ARM systems, we updated and improved an existing incomplete ARM port[3] of S^{2}E, to suit the needs of dynamic analysis of firmware binaries.

To control the execution of code in more detail, S^{2}E provides a powerful plugin interface that allows instrumentation of virtually every aspect of execution. Any emulation event (e.g., translation of a basic block, instruction translation or execution, memory accesses, processor exceptions) can be intercepted by a plugin, which then can modify the execution state according to its needs. This modular architecture let us perform dynamic analysis of firmware behaviour, such as recording and sandboxing memory accesses, performing live migration of subroutines (see Section III-C), symbolically executing specific portion of code as well as detecting vulnerabilities (see Section V).

S^{2}E is connected through three different control interfaces with *Avatar*: the first interface is a GDB debug connection using the GDB serial protocol. *Avatar* is connecting to this interface using a GDB instance controlled via the GDB/MI protocol. This connection is used for fine-grained control over the execution, such as putting breakpoints, single-stepping the execution, and inspecting register values. The second interface is Qemu's Management Protocol (QMP) interface, a JSON-based request-response protocol. Though detailed virtual machine control is possible through this interface, it is currently only used to dynamically change S^{2}E's configuration at run time. This is done by accessing S^{2}E through its Lua interface, which is called from Lua code embedded in the JSON requests. The third interface is a plugin for S^{2}E that is triggered whenever a memory access is performed. This S^{2}E plugin then forwards this request to *Avatar*, which in turn handles the memory access (e.g., sends it to *Avatar*'s plugins), or forwards it to the target.

Even though at the moment the only available emulator back-end is for Qemu/S^{2}E, the emulator interface is generic and allows other emulators to be added easily.

[2]The *Avatar* framework is open-source and available at http://s3.eurecom.fr/tools/avatar.

[3]Our patches have been submitted to the official S^{2}E project and are currently under review for merging.

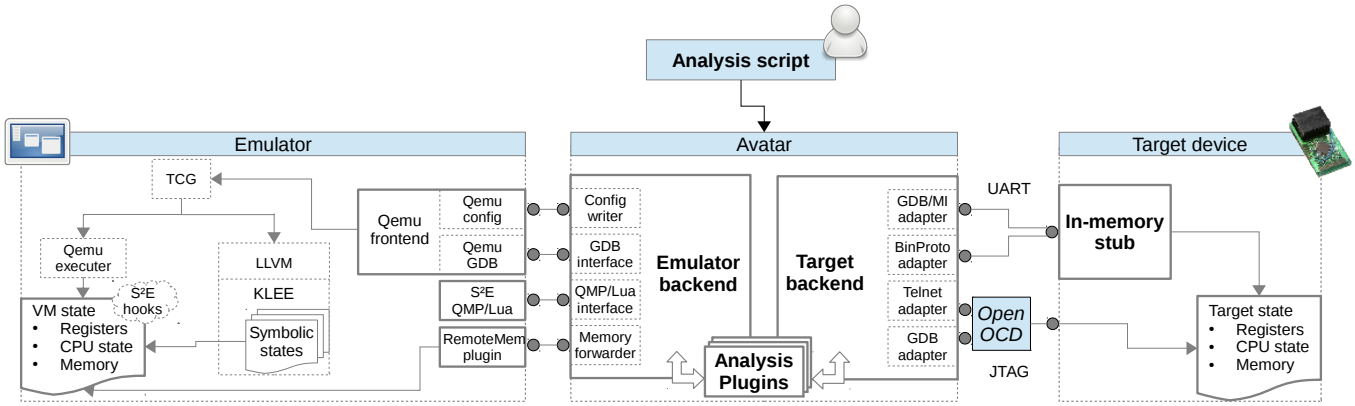

Fig. 1: Overview of *Avatar*.

On the target side, we developed three back-ends:

- A back-end that uses the GDB serial protocol to communicate with GDB servers (e.g., a debugger stub installed on the device or a JTAG GDB server).
- A back-end to support low-level access to the OpenOCD's JTAG debugging interface via a telnet-like protocol.
- A back-end that talks to a custom *Avatar* debugger proxy over an optimized binary protocol (which is more efficient than the verbose protocol used by GDB). This proxy can be installed in an embedded device that lacks debugging hardware support (e.g., no hardware breakpoints) or on which such support was permanently deactivated.

The proper target back-end has to be selected by the user based on the characteristics and the debugging functionalities provided by the hardware of the embedded device. For example, in our experiments we used the OpenOCD back-end to connect to the JTAG debugger of the mobile phone and of the Econotag, while we used the *Avatar* proxy to perform dynamic analysis of the hard drive firmware.

To analyze a firmware, an access to the firmware's device is needed. This can be either a debugging link (e.g., JTAG), a way to load software or a code injection vulnerability. In cases where a debugging stub, for example the GDB stub, is used, an additional communication channel, e.g., an UART, is also needed.

B. Full-Separation Mode

When *Avatar* is first started on a previously unknown firmware, it can be run in what we call *"full-separation mode"*. In this configuration, the entire firmware code is executed in the emulator and the entire (memory) state is kept in the physical device. In other words, for each instruction that is executed by the emulator, the accessed memory addresses are fetched from and written to the real memory of the embedded system. At the same time, interrupts are intercepted by the debugging stub in the physical system and forwarded back to the emulator. Code and memory are perfectly separated, and *Avatar* is responsible to link them together.

Even though this technique is in theory capable of performing dynamic analysis on unknown firmwares, it has several practical limitations. First of all, the execution is very slow. Using a serial debug channel at 38400 Baud, the system can perform around five memory accesses per second, reducing the overall emulation speed to the order of tens instructions per second. Even worse, many physical devices have time-critical sections that need to be executed in a short amount of time or the execution would fail, making the system crash. For example, DRAM initialization, timer accuracy and stability checks belong to this category.

Moreover, tight hardware-polling loops (e.g., UART read-with-timeout) become painfully slow in full separation mode. Finally, regular interrupts (e.g., the clock tick) quickly overload the limited bandwidth between the target system and the emulator.

These limitations make the full separation approach viable only to analyze a limited number of instructions or when the user wants to focus only on particular events in more complex firmwares. For this reason, *Avatar* supports arbitrary context-switching between the emulator and the real device.

C. Context Switching

While it is possible to run the firmware code from beginning to end inside the emulator, sometimes it is more efficient to let the firmware run natively on the target device for a certain amount of time. This allows, for example, to execute the code without any delay until a particular point of interest is reached, skipping through initialization routines that may involve intensive I/O operations or network protocol communications that may need to be performed in real-time. In such cases, it is important to let the target device run the firmware, while still monitoring the execution for regions of code relevant to the current analysis. The ability of *Avatar* to perform arbitrary context switches gives the user the ability to quickly focus her analysis on a particular section of the code, without the drawbacks of emulating the entire firmware execution.

Starting the analysis at specific points of interest: In this case the firmware starts the execution on the physical device and runs natively until a certain pre-defined event occurs (e.g.,

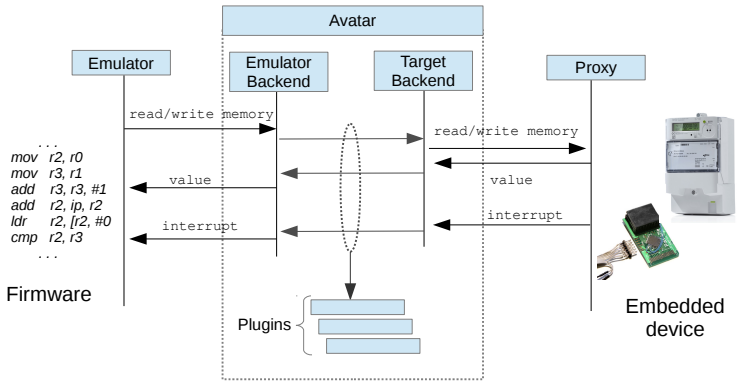

Fig. 2: *Avatar* architecture and message exchange in full separation mode.

a breakpoint is reached or an exception is raised). At this point, the execution on the physical device is frozen and the state (e.g., the content of the CPU registers) is transferred to the emulator, where the execution is resumed. An example of this transition is described in Section VI-C, in which the firmware of a mobile phone baseband chip is executed until the phone receives an SMS, and then transferred by *Avatar* in the emulator to perform further analysis.

Returning execution to the hardware: After the required analysis is performed on the emulator, the execution of the firmware can be transferred back to continue on the real device. In this case, any state kept on the virtual environment is copied back to the physical device. Depending on the user's needs, it is possible to switch again to the emulator at a later stage. This approach is used in Section VI-A, in which the firmware of a hard disk is started inside the emulator and later transferred back to the disk.

D. Interrupts Handling

Software interrupts do not present a problem for our framework, since they are issued by the firmware code and the emulator takes care of calling the corresponding interrupt handler directly. However, as shown in Figure 2, hardware interrupts need to be trapped in the real hardware and forwarded back to the emulator. In this case, the stub in the embedded system receive the interrupt and forwards them to *Avatar*'s target back-end. Finally, using the emulator back-end, *Avatar* suspends the firmware execution and injects the interrupt in the emulator.

Based on the circumstances in which the interrupt is generated, we distinguish three different cases:

- Hardware interrupts that indicate the completion of a task. These interrupts are issued by a device to indicate that a particular task initiated by the code has been completed. For example, the UART *send* interrupt indicates that the send buffer has been successfully transmitted. This type of interrupts is easy to handle because it just needs to be forwarded from the target to the emulator.
- Periodical hardware interrupts, e.g., the timer notifications. These interrupts can be forwarded to the emulator but their frequency needs to be scaled down to the actual execution speed in the emulator. The equivalent number of instructions between two interrupts should be executed in the emulator as it would on the target running in native mode. In our current implementation, an *Avatar* plugin detects periodic interrupts and report their information to the user, who can decide how to handle each class. For example, the user can instruct *Avatar* to drop the clock interrupts on the device and just generate them (at the right frequency) on the emulator, thus saving bandwidth and increasing the analysis performance.
- Hardware interrupts that notify of an external event. For example the *receive* interrupt of an UART indicates that new data on the UART buffer is available. The emulation strategy for those interrupts depends on the frequency of the external event. For events that require previous activity (e.g., a request-response protocol where the response triggers an interrupt) a simple forwarding strategy can be used. For unrelated events that happen very frequently (i.e., where the handler in the emulator cannot process the interrupt in time before the next interrupt is generated) the user can choose if she wants to suppress some of them or to handle the interrupt by migrating the handler itself back to the embedded device (see Section IV)

While the straightforward interrupt forwarding does not present any problem for *Avatar*, when the user needs to tune the framework to handle specific cases (e.g., regular or very frequent interrupts) the stub needs to be able to distinguish between them. Unfortunately, this task is often difficult.

Interrupts de-multiplexing: In a traditional, x86-based, personal computer there is a standard interrupt controller that handles interrupt lines from each device and peripheral. However, on ARM-based systems there are only two interrupt lines directly attached and visible to the processor: `IRQ` and `FIQ`. Because of this embedded devices often use an interrupt multiplexer (or controller) peripheral that is normally included as an hardware block ("IP core") on the same chip. The disadvantage for a user is that at the point where the interrupt vector routine is called, all interrupt signals are still multiplexed together. The driver for a particular interrupt multiplexer will then query the underlying hardware multiplexer to identify

which line was actually triggered and then forward the event to the handler registered for this interrupt.

Now, suppose the user wants to instruct *Avatar* to suppress a particular interrupt on the device (e.g., the timer), while still letting through the ones associated to important hardware events that need to be forwarded to the emulator. In this case, the proxy needs to take a decision based on the interrupt type which is unfortunately not available when the interrupt is received.

In this case, the user needs to disassemble the interrupt vector handler, and follow the code flow until the code of the interrupt controller driver branches into different functions that handle each device's interrupt. At this point, she can specify these program points to *Avatar* that can terminate the interrupt vector's execution and signal to the proxy that an interrupt has been identified. The proxy then sends the interrupt event to *Avatar*. Now the target backend of *Avatar* can suppress a particular interrupt by instructing the proxy to drop the corresponding event.

E. Replaying Hardware Interaction

It is quite common for a firmware to have several sections that require only a limited interaction with dedicated peripherals. In this case, the I/O operations can be recorded by *Avatar* and transparently replayed during the next execution of the firmware.

This allows the user to test the firmware without the bottleneck of the interaction with the physical device. In this mode of operation the firmware itself or parts of it (e.g., applications) can be significantly changed, as long as the order of I/O interactions is not modified. This is a major advantage over resuming a snapshot, which requires the full code path until the snapshot point to be executed to ensure that peripherals are in the state the snapshot expects them to be in.

IV. Overcoming the limits of Full Separation

The techniques introduced in the previous section are enough to perform dynamic analysis on small portions of a firmware code. However, sometimes the internals and behavior of the system are completely unknown. In those cases, it can be very useful to perform the analysis on larger portions of the binary, or, in the extreme case, on the entire firmware.

In this case, the performance of *Avatar* running in full separation mode poses a great limitation to the usability of our framework. To overcome this problem, in this section we present two techniques designed to overcome the limits of full separation by moving part of the code to the physical device and part of the memory to the emulator. This results in a considerable reduction in the number of messages forwarded by *Avatar* between the emulator and the target, and therefore a large improvement in the overall performance of the analysis system.

A. Memory Optimization

Forwarding all memory accesses from the emulator to the target over a limited-bandwidth channel like UART or JTAG incurs in a heavy performance penalty. For example, in our experiments an average of five instructions per second were executed using the GDB stub through a 38400 baud UART connection.

Access type	Read	Write	Cumulative
Code	61,632	-	61,632
Stack & data	646	1,795	64,073
I/O	3,614	2,097	69,784

TABLE I: Number of memory accesses grouped by memory regions for the HDD bootloader.

The reason why memory operations need to be forwarded in the first place is that different embedded systems typically have different mappings of addresses to memory regions. Some of these memory regions are used for code (in RAM, ROM or Flash memories), stack and heap, but one or several regions will be used to access registers of physical peripherals through Memory-Mapped I/O (MMIO). In this case, any I/O operation on those areas is equivalent to sending and receiving data from an external device. If these address ranges are known, the user can configure *Avatar* to keep every read-only memory (such as the code segment) on the emulator. Read-write memory regions can also be marked as local to the emulator, but modifications to them need to be tracked by *Avatar* to be able to transfer those changes to the target at a later context switch. In fact, when an emulator-to-target context switch happens, all modified local memory ("dirty memory") needs to be copied to the target before the execution can resume on the embedded device.

However, in most of the cases the user does not know a priori which area of memory is assigned to I/O. For this reason, *Avatar* includes an automated memory optimization plugin that monitors the execution in the emulator and automatically identifies the regions that do not require access to the hardware. This includes the stack (easily identified by the execution of stack-related operations) and the code segment (identified by the values of the program counter). For any other area, *Avatar* starts by forwarding the read and write operations to the target device. It then keeps track of the values that are returned and applies a simple heuristic: if the target always returns the value that was previously written by the firmware code (or if it always returns the same value and it is never written by the firmware) then it is probably not assigned to a memory mapped device.

Table I shows an example of how many memory accesses could be saved by keeping memory regions local to the emulator: transferring the code region to the emulator would save 61,632 memory accesses (88%). Moving the stack and data region in local memory as well would save 64,073 memory accesses (92%). Only the I/O accesses cannot be moved to the emulator's memory.

B. Selective Code Migration

So far, we assumed that the firmware is either running entirely inside the emulator, or entirely on the embedded device. The user can instruct *Avatar* to switch from one mode to the other when certain conditions are met, but such context switches are time consuming.

In this section we present a fine-grained solution that allows the user to migrate only parts of the firmware code back to the target. This technique allows to overcome two limitations of the full-separation mode. Some code blocks need to be executed atomically, for example when there are timing constraints on the code. We will describe such a case in Section VI-A, where we encountered a function that read the timer twice and waited for the difference to be below a certain limit. Another example is when delays introduced by *Avatar* would lead the target in an invalid state. We encountered such a case during the DRAM initialization of the HDD, as shown in Section VI-A).

The second limitation addressed by selective code migration is related to the analysis performance. In fact, certain functions (e.g., polling loops and interrupt handlers) can be executed significantly faster when run natively on the target.

In the current *Avatar* prototype, code migration is supported at a function level. In this case, the code can be copied to its location in the target's memory without modification. Its exit points are then replaced by breakpoints, and the virtual machine register state is transferred from the emulator to the target. The execution is resumed on the target until one of the exit breakpoints is triggered, and at that point the state is transferred back to the emulator. This transition is much faster than a complete context switch, since *Avatar* only needs to transfer few bytes and not the entire content of the memory.

Even though this simple technique is enough to circumvent critical code regions in several real world scenarios, it neglects some difficulties that may affect code migration. First, the code may read or write arbitrary memory locations associated, for example, with global variables. *Avatar* keeps track of those locations, copy their content over to the target before the execution, and copy written locations back after the execution. Second, the code may use instructions that change the control flow in unforeseen ways, like software interrupts, processor mode changes, and indirect jumps.

Our framework prototype addresses these issues by performing an on-the-fly static analysis. When a function is selected for code migration, *Avatar* disassembles its code using the `llvm-mc` disassembler. The result is then analyzed to identify critical instructions. In this way, we can predict memory accesses outside the function stack, compute the control flow of the code and verify that no instructions can escape from this computed control flow. As we describe in Section VI, this technique is sufficient to migrate small, atomic functions. However, we plan to extend the capabilities of the code migration system to apply transformations to the code. On the one hand, those transformations will allow to ensure that instructions which are not statically verifiable (e.g., indirect jumps) will not escape the proxy's sandbox. On the other hand, it can be used to track memory accesses, so that only the modified ("dirty") part of the state needs to be copied back from the target to the emulator when a context switch happens. Those critical instructions will be replaced with instrumentation code that calls functions in proxy, which will handle them in a safe way.

V. Extending Avatar

Avatar's architecture is designed to be modular and its base framework can be easily customized to fit different analysis scenarios. We chose S^2E as default *Avatar* emulator back-end because it offers many hooks and manipulation facilities on top of QEMU which facilitates the development of custom dynamic analysis plugins.

In this section, we show an example of an *Avatar* extension: we built upon its core capabilities to support selective symbolic execution. For this we add several features and plugins to the ARM port of S^2E. Moreover, we believe the symbolic execution engine provides a super-set of the capabilities needed to implement taint analysis, even though a targeted plugin could be needed to perform concrete data tracking and taint analysis in a more lightweight way.

In the rest of this section we describe the technique *Avatar* employs to fully exploit the symbolic engine of S^2E and perform selective symbolic execution on unmodified portions of firmware blobs. Moreover, we show how we use our extended version of S^2E in *Avatar* to dynamically detect potential control flow corruption vulnerabilities by injecting and tracking symbolic inputs.

A. Injecting Symbolic Values Into the Firmware's Execution Flow

In the field of program testing, symbolic execution is a technique employed to improve code coverage by using symbols as input data (instead of concrete values) and keeping track of constraints upon their manipulation or comparison (c.f. [51]). The result of symbolic evaluation is an execution tree, where each path is a possible execution state that can be reached by satisfying the constraints associated to each symbolic value.

S^2E further develops this concept by performing selective symbolic execution, i.e., by restricting the area of symbolic execution to specific code portions and treating only specific input data as symbolic [15]. This greatly helps to speedup the analysis process (as symbolic execution of code results in significant slowdowns) and to drive the exhaustive symbolic exploration into selected regions of code. This process requires *Avatar* to control the introduction of symbolic values into S^2E, in place of existing real values.

The remote memory interface between S^2E and *Avatar*, as introduced in Section III, ensures that only concrete values reach the real hardware through *Avatar*. Symbolic values remain therefore confined to the emulation domain. If a symbolic value is about to be written to the target hardware, the remote memory interface in S^2E performs a forced concretization before forwarding it. Such symbolic value concretizations happen in two stages. First, all the constraints associated with the value are retrieved and evaluated by the integrated SAT-solver. Second, a single example value which satisfies all the constraints is forwarded to *Avatar* to be written on the target.

On the one hand, making *Avatar* handle only concrete values leaves it as a controller with a simpler external view of S^2E and avoids having to keep track of execution paths and paths conditions twice. On the other hand, this choice brings the minor drawback that *Avatar* has no direct control

on symbolic execution, which is instead under the control of S^{2}E/KLEE.

We designed a simple plugin for detecting arbitrary execution conditions. It relies on the following heuristics as signs of possibly exploitable conditions:

- a symbolic address being used as the target of a load or store instruction,
- a symbolic address being leaked into the program counter (e.g., as the target of a branch),
- a symbolic address being moved into the stack pointer register.

In order to selectively mark some input data as symbolic, two different approaches can be taken: either modify the binary code (or the source code, if available) to inject custom instructions into the firmware, or dynamically instrument the emulation environment to specify the scope of symbolic analysis at run-time. The first approach requires some high-level knowledge of the firmware under analysis (e.g., access to source code) and the guarantee that injecting custom instructions into firmware code would not affect its behavior. Examples include the Android Dalvik VM, whose source code can be modified and rebuilt to enable transparent analysis of pristine Java bytecode with S^{2}E [36].

Since we did not want to limit *Avatar* to this scenario, we decided to follow the second approach, which requires to extend the symbolic engine and the *Avatar* framework. Such extensions should know when symbolic execution has to be triggered and where symbolic values should be injected.

This choice leads to two major advantages:

- *Firmware Integrity*
  The binary code is emulated as-is, without injecting custom opcodes or performing recompilation. This guarantees that the emulated code adheres to the original firmware behavior (i.e., no side-effects or bugs are introduced by the intermediate toolchain)
- *Programmatic Annotation*
  The control and data flow of firmware emulation can be manipulated and annotated with symbolic meta-data in an imperative way. A high-level language (Lua) is used to dynamically script and interact with current emulation environment, as well as introducing and tracing symbolic meta-data.

For this we first completed the port of S^{2}E to the ARM architecture in order to have complete symbolic execution capabilities, then we ported the Annotation plugin to the ARM architecture. The Annotation plugin lets the user specify a trigger event (e.g., a call/return to a specific subroutine or the execution of code at a specific address), and a Lua function to be executed upon the event. A simple API is then provided to allow for manipulation of the S^{2}E emulation environment directly from the Lua code. *Avatar* provides direct channels to dynamically control the emulation flow via QMP command messages. These channels can also be used to inject Lua code at run-time, in order to dynamically generate annotations which depend on the current emulation flow and inject them back into S^{2}E. Once symbolic values are introduced in the execution flow, S^{2}E tracks them and propagates the constraints.

Symbolic analysis via Lua annotations is intended to be used as a tool for late stage analysis, typically to ease the discovery of flaws in logic-handling code, with hand-made Lua analysis code directly provided by the user. It can be employed in both full separation mode and context switching, as soon as code execution can be safely moved to the emulator (e.g., outside of raw I/O setup routines, sensors polling). This normally happens after an initial analysis has been done with *Avatar* to detect interesting code and memory mappings.

A similar non-intrusive approach has already been used in a x86-specific context, to test and reverse-engineer the Windows driver of a network card [14]. To the best of our knowledge, however, this technique has never been applied before to embedded devices. In the context of firmware security testing, annotations can be used in a broad range of scenarios. In Section VI, we present how we applied this technique to different technologies and devices, to perform dynamic analysis of widespread embedded systems such as hard drives, GSM phones, and wireless sensors.

B. Symbolically Detecting Arbitrary Execution Conditions

When dealing with modern operating systems, an incorrect behavior in a user-space program is often detected because an invalid operation is performed by the program itself. Such operations can be, for example, an unauthorized access to a memory page, or the access to a page that is not mapped in memory. In those cases, the kernel would catch the wrong behavior and terminate the program, optionally triggering some analysis tools to register the event and collect further information that can later be used to identify and debug the problem. Moreover, thanks to the wide range of exploit mitigation techniques in place today (DEP, canaries, sandboxing and more), the system is often able to detect the most common invalid operations performed by userspace processes.

When dealing with embedded systems, however, detecting misbehavior in firmware code can be more difficult. The observable symptoms are not always directly pinpointed to some specific portion of code. For example, many firmware are designed for devices without a Memory Management Unit (MMU) or Memory Protection Unit (MPU) or are just not using them. In such a context, incorrect memory accesses often result in subtle data corruption which sometimes leads to erratic behaviors and rare software faults, such as random events triggering, UI glitches, system lock or slowdown [18]. For this reason, it is common for embedded devices to have a hardware watchdog in charge of resetting the device execution in case of any erratic behavior, e.g., a missed reply to timed watchdog probes.

For these reasons, detecting incorrect execution inside the emulation is easier when some OS support can be used for co-operation (e.g., a *Blue Screen Of Death* interceptor for Windows kernel bugs is implemented in S^{2}E). On the other hand, catching such conditions during the emulation of an embedded device firmware is bound to many system-specific constraints, and require additional knowledge about the internal details of the firmware under analysis.

However, *Avatar* does not rely on the knowledge of any specific operating system or the fact that a MMU is used. Instead, it aims at detecting a larger range of potentially critical situations which may result in control flow hijacking of firmware code, by using a technique similar to the one employed by AEG [6].

All three conditions may lead to false positives, when the variable is symbolic but strongly constrained. Therefore, once such a condition is detected the constraints imposed on the symbolic variables must be analyzed: the less constrained is the result, the higher is the chance of control flow corruption. Intuitively, if the constraints are very loose (e.g., a symbolic program counter without an upper bound) then the attacker may obtain enough control on the code to easily exploit the behavior. In addition to this, tight constraints are sometimes encountered in legitimate cases (e.g., access to an array with a symbolic but constrained index such as with a jump table), and are not relevant for the purpose of security analysis.

When an interesting execution path is detected by the above heuristic, the state associated to the faulty operation is recorded and the emulation is terminated. At this point a test-case with an example input to reach this state is generated, and the constraints associated with each symbolic value are stored to be checked for false positives (i.e., values too strictly bound).

Automatically telling normal constraints apart from those that are a sign of a vulnerability is a complex task. In fact it would require knowledge of the program semantics that were lost during compilation (e.g., array boundaries). Such knowledge could be extracted from the source code if it is available, or might be extrapolated from binary artifacts in the executable itself or the build environment. In such cases, specific constraints could be fed into *Avatar* by writing appropriate plugins to parse them, for example by scanning debug symbols in a non-stripped firmware (e.g., a DWARF parser for ELF firmwares) or by reading other similar symbols information.

Finally, *Avatar* could highly benefit from a tighter coupling with a dynamic data excavator, helping to reverse engineer firmware data structures [17]. In particular, the heuristic proposed in Howard [52] for recovering data structures by observing access patterns under several execution cycles could be easily imported into the *Avatar* framework. Both tools perform binary instrumentation on top of QEMU dynamic translation and make use of a symbolic engine to expand the analyzed code coverage area.

C. Limitations of state synchronization

Our current implementation of the synchronization between device state and emulator state works well in general, but is difficult in some special cases.

First it is difficult to handle DMA memory accesses in our current model. For example, the firmware can send a memory address to a peripheral and request data to be written there. The peripheral will then notify the firmware of the request's completion using an interrupt. Since *Avatar* does not know about this protocol between firmware and peripheral, it will not know which memory regions have been changed. On newer ARM architectures with caches, *data synchronization barrier* or *cache invalidation* instructions might be taken as hint that some memory region has been changed by DMA.

Second, if code is executed on the device, *Avatar* is currently incapable of detecting which regions have been modified. In consequence, whenever memory accesses of the code run on the device are not predictable by static analysis, we need to transfer the whole memory of the device back to the emulator on a device-to-emulator state switch. We plan to address this issue by using checksumming to detect memory region changes and minimize transferred data by identifying smallest changed regions through binary search.

Third, when *Avatar* performs symbolic execution, symbolic values are confined to the emulator. In case that a symbolic value needs to be concretized and sent to the device, a strategy is needed to keep track of the different states and I/O interactions that were required to put the device in that state. This can be performed reliably by restarting the device and replaying I/O accesses. While this solution ensures full consistency, it is rather slow.

VI. Evaluation

In this section we present three case studies to demonstrate the capabilities of the *Avatar* framework on three different real world embedded systems. These three examples by no means cover all the possible scenarios in which *Avatar* can be applied. Our goal was to realize a flexible framework that a user can use to perform a wide range of dynamic analysis on known and unknown firmware images.

As many other security tools (such as a disassembler or an emulator), *Avatar* requires to be configured and tuned for each situation. In this section, we try to emphasize this process, in order to show all the steps a user would follow to successfully perform the analysis and reach her goal. In particular, we will discuss how different *Avatar* configurations and optimization techniques affected the performance of the analysis and the success of the emulation.

Not all the devices we tested were equipped with a debug interface, and the amount of available documentation varied considerably between them. In each case, human intervention was required to determine appropriate points where to hook execution and portions of code to be analyzed, incrementally building the knowledge-base on each firmware in an iterative way. A summary of the main characteristics of each device and of the goal of our analysis is shown in Table II.

A. Analysis of the Mask ROM Bootloader of a Hard Disk Drive

Our first case study is the analysis of a masked ROM bootloader and the first part of the secondary bootloader of a hard disk drive.

The hard disk we used in our experiment is a commercial-off-the-shelf SATA drive from a major hard disk manufacturer. It contains an ARM 966 processor (that implements the ARMv5 instruction set), an on-chip ROM memory which contains the masked ROM bootloader and some library functions, an external serial flash that is connected over the SPI bus to the processor, a dynamic memory (SDRAM) controller, a serial port accessible through the master/slave jumpers, and

	Target device	Manufacturer and model	System-on-Chip	CPU	Debug access	Analyzed code	Scope of analysis
Experiment VI-A	Hard disk	*undisclosed*	unknown	ARM966	Serial port	Bootloader	Backdoor detection
Experiment VI-B	ZigBee sensor	Redwire Econotag	MC13224	ARM7TDMI	JTAG	ZigBee stack	Vulnerability discovery
Experiment VI-C	GSM phone	Motorola C118	TI Calypso	ARM7TDMI	JTAG	SMS decoding	Reverse engineering

TABLE II: Comparison of experiments described in Section VI.

Fig. 3: The disk drive used for experiments. The disk is connected to a SATA (Data+Power) to USB interface (black box on the right) and its serial port is connected to a TTL-serial to USB converter (not shown) via the 3 wires that can be seen on the right.

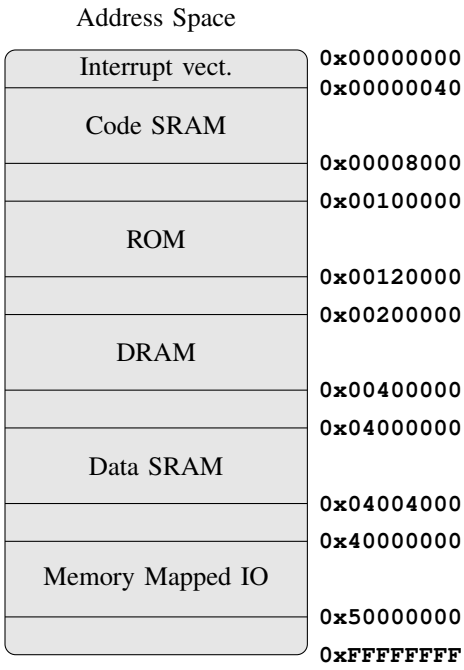

Fig. 4: Hard drive memory layout.

some other custom hardware that is necessary for the drive's operation. The drive is equipped with a JTAG connection, but unfortunately the debugging features were disabled in our device. The hard drive's memory layout is summarized in Figure 4.

The stage-0 bootloader executed from mask ROM is normally used to load the next bootloader stage from a SPI-attached flash memory. However, a debug mode is known to be reachable over the serial port, with a handful of commands available for flashing purposes. Our first goal was to inject the *Avatar* stub through this channel to take over the booting process, and later use our framework for deeper analysis of possible hidden features (e.g., backdoors reachable via the UART).

The first experiment we performed consisted of loading the *Avatar* stub on the drive controller and run the bootloader's firmware in full separation mode. This mimics what a user with no previous knowledge of the system would do in the beginning. In full separation mode, all memory accesses were forwarded through the *Avatar* binary protocol over the serial port connection to the stub and executed on the hard drive, while the code was interpreted by S^2E. Because of the limited capacity of the serial connection, and the very intensive I/O performed at the beginning of the loader (to read the next stage from the flash chip), only few instructions per second were emulated by the system. After 24 hours of execution without even reaching the first bootloader menu, we aborted the experiment.

In the second experiment we kept the same setting, but we used the memory optimization plugin to automatically detect the code and the stack memory regions and mark them as local to the emulator. This change was enough to reach the bootloader menu after approximately eight hours of emulation. Though considerably faster than in the first experiment, the overhead was still unacceptable for this kind of analysis.

Since the bottleneck of the process was the multiple read operations performed by the firmware to load the second stage, we configured *Avatar* to replay the hardware interaction from disk, without forwarding the request to the real hardware. In particular, we used the trace of the communication with the flash memory from the second experiment to extract the content of the flash memory, and dump it into a file. Once the read operations were performed locally in the emulator, the bootloader menu was reached in less than four minutes.

At this point, we reached an acceptable working configuration. In the next experiment, we show how *Avatar* can be used in conjunction with the symbolic execution of S^2E to automatically analyze the communication protocol of the hard drive's bootloader and detect any hidden backdoor in it.

DS	Use a minimal version of the Motorola S-Record binary data format to transmit data to the device
AP <addr>	Set the value of the address pointer from the parameter passed as hexadecimal number. The address pointer provides the address for the read, write and execute commands.
WT <data>	Write a byte value at the address pointer. The address pointer is incremented by this operation. The reply of this command depends on the current terminal echo state.
RD	Read a byte from the memory pointed to by the address pointer. The address pointer is incremented by this operation. The reply of this command depends on the current terminal echo state.
GO	Execute the code pointed to by the address pointer. The code is called as a function with no parameters, to execute Thumb code one needs to specify the code's address + 1.
TE	Switch the terminal echo state. The terminal echo state controls the verbosity of the read and write commands.
BR <divisor>	Set the serial port baud rate. The parameter is the value that will be written in the baud rate register, for example "A2" will set a baudrate of 38400.
BT	Resume execution with the firmware loaded from flash.
WW	Erase a word (4 bytes) at the address pointer and increment address pointer.
?	Print the help menu showing these commands.

TABLE III: Mask ROM bootloader commands of the hard drive. In the left column you can see the output of the help menu that is printed by the bootloader. In the right column a description obtained by reverse engineering with symbolic execution is given.

We configured *Avatar* to execute the hard drive's bootloader until the menu was loaded, and then replace all data read from the serial port register by symbolic values. As a result, S^2E started exploring all possible code paths related to the user input. This way, we were able to discover all possible input commands, either legitimate or hidden (which may be considered backdoors), that could be used to execute arbitrary code by using S^2E to track when symbolic values were used as address and value of a memory write, and when the program counter would become symbolic. With similar methodologies, a user could use symbolic execution to automatically discover backdoors or undocumented commands in input parsers and communication protocols.

In order to conduct a larger verification of the firmware input handler, we were also able to recover all the accepted commands and verify their semantics. Since the menu offered a simple online `help` to list all the available commands, we could demonstrate that *Avatar* was indeed able to automatically detect each and all of them (the complete list is reported in Table III). In this particular device, we verified that no hidden commands are interpreted by the firmware and that a subset of the commands can be used to make arbitrary memory modifications or execute code on the controller, as documented.

However, we found that the actual protocol (as extracted by symbolic analysis) is much looser than what is specified in the `help` menu. For example the argument of the 'AP' command

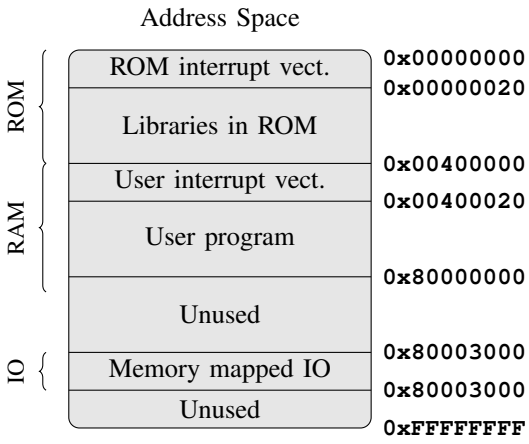

Fig. 5: Econotag memory layout (respective scales not respected).

can be separated by any character from the command, not only spaces. It is also possible to enter arbitrarily long numbers as arguments, where only the last 8 digits are actually taken into account by the firmware code.

After the analysis of the first stage was completed, we tried to move to the emulation of the second stage bootloader. At one point, in what turned out to be the initialization of the DRAM, the execution got stuck: the proxy on the hard drive would not respond any more, and the whole device seemed to have crashed. Our guess was that the initialization writes the DRAM timings and needs to be performed atomically. Since we already knew the exact line of the crash from the execution trace, it was easy to locate the responsible code, isolate the corresponding function, and instruct *Avatar* to push its code back to be executed natively on the hard drive.

In a similar manner, we had to mark few other functions to be migrated to the real hardware. One example is the timer routine, which was reading the timer value twice and then checked that the difference was below a certain threshold (most probably to ensure that the timer read had not been subject to jitter). Using this technique, in few iterations we managed to arrive at the final *Avatar* configuration that allowed us to emulate the first and second stages up to the point in which the disk would start loading the actual operating system from the disk's platters.

B. Finding Vulnerabilities Into a Commercial Zigbee Device

The Econotag, shown in Figure 6, is an all-in-one device for experimenting with low power wireless protocols based on the IEEE 802.15.4 standard [32], such as Zigbee or 6lowpan [43]. It is built around the MC13224v System on a Chip from Freescale. The MC13224v [47] is built upon an ARM7TDMI microcontroller, includes several memories, peripherals and has an integrated IEEE 802.15.4 compatible radio transceiver. As it can be seen in Figure 5, the device includes 96KB of RAM memory, 80 KB of ROM and a serial Flash for storing data. The ROM memory contains drivers for several peripherals as well as one to control the radio, known as *MACA* (MAC Accelerator), which allows to use the dedicated hardware logic

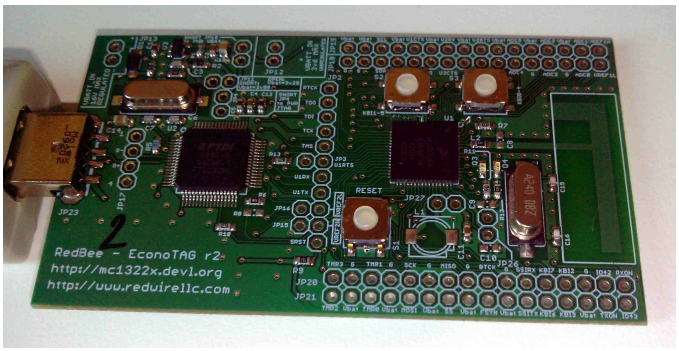

Fig. 6: The Econotag device. From left to right: the USB connector, serial and JTAG to USB converter (FTDI), Freescale MC13224v controller and the PCB 2.4 GHz antenna.

supporting radio communications (e.g., automated ACK and CRC computation).

The goal of this experiment is to detect vulnerabilities in the code that process incoming packets. For this purpose, we use two Econotag devices and a program from the Freescale demonstration kit that simulates a wireless serial connection (wireless UART [26]) using the *Simple MAC* (SMAC [27]) proprietary MAC layer network stack. The program is essentially receiving characters from its UART and transmitting them as radio packets as well as forwarding the characters received on the radio side to its serial port. Two such devices communicating together essentially simulate a wireless serial connection.

The data received from the radio is buffered before being sent to the serial port. For demonstration purposes, we artificially modified this buffer management to insert a vulnerability: a simple stack-based buffer overflow. We then compiled this program for the Econotag and installed it on both devices.

Avatar was configured to let the firmware run natively until the communication between the two devices started. At this point, *Avatar* was instructed to perform a context switch to move the run-time state (registers and data memory) of one of the devices to the emulator. At this point, the execution proceeded in full separation mode inside the emulator using the code loaded in ROM memory (extracted from a previous dump), and the code loaded in RAM memory (taken from the application). Every I/O access was forwarded to the physical device through the JTAG connection.

The emulator was also configured to perform symbolic execution. For this purpose, we used *annotations* to mark the buffer that contains the received packet data as symbolic. Then, we employed a state selection strategy to choose symbolic states which maximize the code coverage, leading to a thorough analysis of the function.

On the first instruction that uses symbolic values in the buffer, S^2E would switch from concrete to symbolic execution mode. Execution will fork states when, for example, conditional branches that depend on such symbolic values are evaluated. After exploring 564 states, and within less than a minute of symbolic execution, our simple *arbitrary execution detection module* detected that an unconstrained

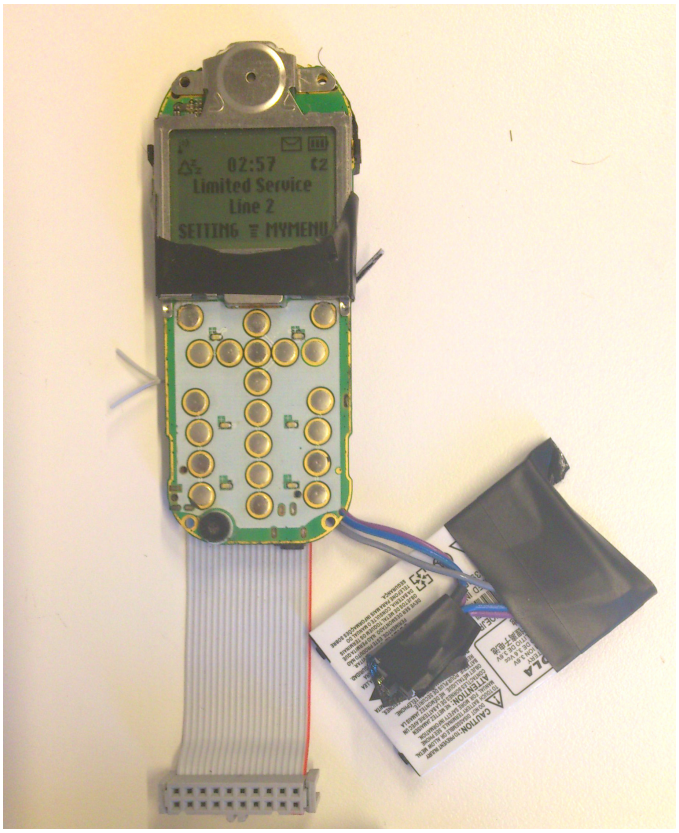

Fig. 7: The Motorola C118. The clip-on battery (on the right) has been wired to the corresponding power pins, while the ribbon cable is connected to the JTAG pads reachable on the back (not shown).

symbolic value was used as a return address. This confirmed the detection of the vulnerability and also provided an example of payload that triggers the vulnerability.

We also used *Avatar* to exhaustively explore all possible states of this function on a program without the injected vulnerability, and confirmed the absence of control flow corruption vulnerabilities that could be triggered by a network packet (that our simple arbitrary execution detection module could detect).

C. Manipulating the GSM Network Stack of a Common Feature Phone

Our final test-case is centered on the analysis of the firmware of a common GSM feature phone. In contrast with most recent and advanced mobile phones and smartphones, feature phones are characterized by having one single embedded processor for both the network stack (i.e., GSM baseband capabilities) and the Human-to-Machine Interface (HMI: comprising the main Graphical User Interface, advanced phone services, and miscellaneous applications). As such, there is no clear code separation between different firmware sections. On these phones, typically a real-time kernel takes care of scheduling all the tasks for the processes currently in execution. These are executed in the same context and have shared access to the whole physical memory as well as memory-mapped I/O.

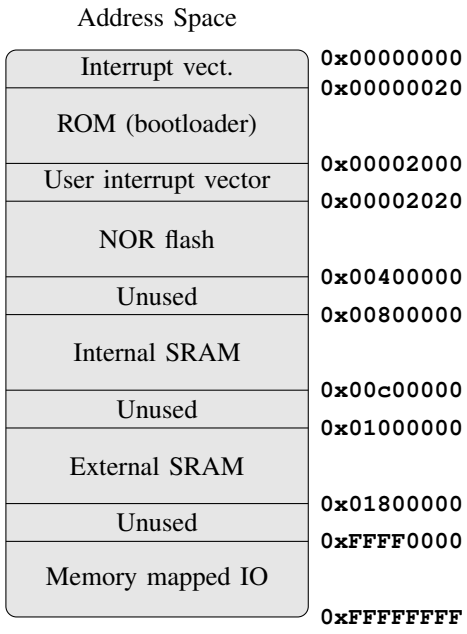

Fig. 8: Motorola C118 memory layout (respective scales not respected).

GSM baseband stacks have already been shown to have a large potentially exploitable attack surface [56]. Those stacks are developed by few companies worldwide and have many legacy parts which were not written with security in mind, and in particular were not considering attacks coming from the GSM infrastructure [57].

For our experiment, we used a Motorola C118, which is a re-branded version of the Compal E88 board also found in other Motorola feature phones. This board makes use of the *Texas Instruments "Calypso"* digital baseband, which is composed of a mask-ROM, a DSP for GSM signal decoding, and a single ARM7TDMI processor. It also includes several peripherals such as an RTC clock, a PWM generator for controlling the lights and buzzer as well as a memory mapped UART as shown in Figure 8. Some board models have JTAG and UART ports available, which are from time to time left enabled by manufacturers to simplify servicing devices. In our case, we gained access to the JTAG port and used an adapter to bridge communication between *Avatar* and the hardware, as shown in Figure 7.

Some specification documents on the *Calypso* chipset have been leaked in the past, leading to the creation of home-brew phone OS that could be run on such boards. As part of the Osmocom-BB project, most of the platform has been reversed and documented, and it is now possible to run a free open-source software GSM stack on it [2]. However, we conducted our experiments on the original Motorola firmware, in order to assess the baseband code of an unmodified phone. Moreover, as the GSM network code is provided as a library by the baseband manufacturer, there is an higher chance that flaws affecting the library code would also be present in a broader range of phones using baseband chips from that same vendor.

The phone has a first-stage bootloader executed on hardware reset, which can be used to re-flash the firmware. After phone setup, execution continues to the main firmware, which is mainly composed of the Nucleus RTOS, the TI network stack library, and of third-party code to manage the user interface. The phone bootloader can be analyzed using *Avatar* in a similar way as the one already described for the hard disk in Section VI-A to discover flashing commands, hidden menus and possible backdoors. However, the bootloader revealed itself to be simpler than the hard drive one, supporting only a UART command to trigger firmware flashing and executing the flashed firmware, or continuing execution after a timeout expiration.

For this reason, we focused on the analysis of the GSM network stack, and in particular on the routines dedicated to SMS decoding. It has already been shown in the past how maliciously crafted SMS can cause misbehavior, ranging from UI issues to phone crashes [44]. However, due to the lack of a dynamic analysis platform to analyze embedded devices, previous studies relied on blind SMS fuzzing. Our experiment aims at improving the effectiveness of SMS fuzzing to detect remotely exploitable execution paths.

In this scenario, *Avatar* was configured to start the execution of the firmware on the real device, and switch to the emulator once the code reached the SMS receiving state (e.g., by sending a legitimate SMS to it through the GSM network). *Avatar* was then used to selectively emulate and symbolically explore the decoding routines. As a result of this exploration, a user is able to detect faulty conditions, to determine code coverage due to different inputs and to recover precise input constraints to drive the firmware execution into interesting areas.

In this context, *Avatar* uses the JTAG connection to stop the execution on the target and later perform all synchronization steps between the emulator and the target. All memory and I/O accesses through JTAG are traced by *Avatar* to let the user identify address mappings. When the phone reaches the SMS receiving state, a target-to-emulator context switch happens and the phone's state is transferred into S^2E. Using address mapping information previously recovered through *Avatar*, just the relevant memory is moved into S^2E (e.g., portions of code and the execution stack), while remaining memory is kept on the target and forwarded on-the-fly by *Avatar* (e.g., I/O regions). On this device, no selective code migration was required.

Using this *Avatar* configuration, the SMS payload can be intercepted in memory and marked as symbolic by employing the techniques shown in Section V. In particular, we wrote Annotation functions to be triggered before entering the decoding routines and we then proceeded to selectively mark some bytes of their input arguments as symbolic. The S^2E plug-in for Arbitrary Execution Detection has been employed to isolate interesting vulnerable cases, while other execution paths were killed upon reaching the end of the decoding function.

The symbolic execution experiments have been performed over several days, with the ones with larger number of symbolic inputs taking up to 10 hours before filling up 60 GB of available memory. In such case, we observed more than 120,000 states being spawned according to different constraints solving. Unfortunately, and contrary to the other experiments,

the GSM network stack proved to be way too complex to be symbolically analyzed without prior knowledge on the high-level structure of the code. The analysis was clobbered by an explosion of possible states due to many forks happening in pointer-manipulating loops. *Avatar* was able to symbolically explore 42 subroutines executed during SMS decoding, without detecting any exploitable conditions. However, it was able to highlight several situations of user-controlled memory load, which were unfortunately too strictly constrained to be exploited, as discussed in Section V-B.

State explosion is a well-known limitation of symbolic execution. To mitigate the problem, a user may need to define heuristics to avoid an excessive resource consumption. This could be done, for example, by employing more aggressive state selectors to enhance code coverage, and actively prune states by looking at loops invariants [50]. However, this optimizations are outside the scope of our paper. The objective of our experiments are, in fact, limited to prove that *Avatar* can be used to perform dynamic analysis of complex firmware of embedded devices.

VII. Related work

The importance of porting dynamic analysis techniques to different platforms has been discussed by Li and Wang [41], who proposed a set of tools built on top of IDA Pro and the REIL Intermediate Language to perform symbolic execution in a portable way.

However, embedded systems have long been recognized to be a difficult target for debugging and dynamic analysis. SymDrive [48] presents a technique based on symbolic execution to test Linux and FreeBSD device drivers without their device present. However, by replacing every input with a symbolic value, this approach is hard to scale and would suffer of state explosion on any real world firmware. In [14], Chipounov and Candea present REVNIC, a tool based on S^2E [15] that helps to reverse engineer network device drivers. As a case study the authors port a Windows device driver for a common network card to a different Operating System. While the presented approach is interesting, it relies on the presence (and extension) of the emulated device and PCI bus in QEMU. Instead, *Avatar* is hardware agnostic, as it does not need to know how peripherals are connected, mapped and accessed. Instead I/O can be simply forwarded to the real target and I/O related code directly executed there.

Cui et al., adopted *software symbiotes* [20], an on-device binary instrumentation to automatically insert hooks in embedded firmwares. Their solution allows to insert pieces of code that can be used to interact with the original firmware. However, while this allows some analysis (like tracing), performing advanced dynamic analysis often requires to be able to run the firmware code inside an emulator.

Dynamic analysis based on virtualization has already been proposed in the past [37], also in embedded systems contexts [40], [31]. However, *Avatar* aims at overcoming many of the limitations of pure-virtualization systems, by providing an hybrid system where code execution can be transferred back and forth between the device and an emulator, as well as a full framework to orchestrate all the analysis steps.

The state migration technique employed by *Avatar* is highly influenced by existing solutions been used to improve the performance during hot-migration of virtual machines. In particular, our approach is a simplified version of the one proposed by Clark et al. [16], where *Avatar* is the arbiter of a *managed migration*, which can either happen in a single *stop-and-copy phase* (as in full-separation mode) or in an event-driven *pull-phase* (during context switching).

The "security by obscurity" approach is still relevant among embedded systems manufacturers and has lead in the past to the discovery of major weaknesses in commonly deployed technologies [45]. We believe that *Avatar* represents a flexible solution to provide a symbolic analysis environment which can greatly speed-up such blackbox analysis cases, aiming at automatically reverse engineer input formats [21], [9] and detect hidden data structures [53]. In the past, backdoors and insecure firmware update facilities were found into embedded systems, often disguised into other standard interfaces such as Printer Job Language updates for HP printers [19]. In our experiments we showed how *Avatar* can be used to actively look for such backdoors, by symbolically executing input parsing routines.

Davidson et al. [22] present a tool to perform symbolic execution of embedded firmware for MSP430-based devices. Like *Avatar*, this tool is based on the KLEE symbolic execution engine. However, it relies on firmware's source code as well as on documented SoCs, peripherals mapping, or on a simple emulation layer for them, all of those are rarely available for commercial devices.

Delugre [23] reports on the techniques that were used to reverse engineer the firmware of a PCI network card, and to develop a backdoored firmware. For this purpose, QEMU was adapted to emulate the firmware and to forward IO access to the device. However, this was limited by bad performance. We have seen similar performance blockers when using *Avatar* in full separation mode, but the ability to perform memory optimization and push back code to the physical device allow *Avatar* to overcome such limitations.

Dedicated hardware support can provide a very good solution to improve efficiency of debugging, improving significantly the ability to replay events and system status. In [59] Xu et al., presents an hardware architecture for recording precise events and replay them during debugging sessions. For this purpose custom hardware logs memory and taps on several important internal features (e.g., cache lines). Simpler systems also exist, like In-Circuit Emulators [58], which replace the CPU core by an emulated CPU which can then directly interact with hardware peripherals. While *Avatar* could make use of such features, it also aims at enabling analysis on devices without such dedicated hardware support.

VIII. Conclusion

This paper introduced *Avatar*, a new framework for dynamic analysis of embedded devices' firmwares. *Avatar* enables the execution of firmware code in an analysis-friendly emulator by forwarding memory access to the real device. This allows to analyze firmwares that rely on completely unknown peripherals.

Avatar proved to be capable of acceptable performances and flexibility in three real-word tests, performed on a variety of target devices and with different goals. It was successfully used across these three scenarios, which included a common reverse engineering task, a vulnerability discovery and a hard-coded backdoor detection.

Future work will consist in integrating better analysis techniques with avatar to improve its bug detection rate. For example, augmenting *Avatar* with techniques like those used in Howard [52] would allow to recover memory structures and therefore improve bug detection, while other techniques as used in AEG [6] could be applicable as well. Another area where significant improvements can be achieved is in providing improved state exploration heuristics, that lead to better coverage or to the analysis of more error prone code [30].

Finally, *Avatar* has been tested on ARM embedded systems and could easily support x86 targets, but could be ported with reasonable effort to a wider set of architectures supported by QEMU such as MIPS and PowerPC, in order to analyze many other devices.

ACKNOWLEDGMENTS

Authors would like to thank Pascal Sachs and Luka Malisa that worked on an early prototype of the system, and Lucian Cojocar for his helpful comments on the current version of *Avatar*. The research leading to these results was partially funded by the European Union Seventh Framework Programme (contract Nr 257007 and project FP7-SEC-285477-CRISALIS).

REFERENCES

[1] Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/.

[2] OsmocomBB. http://bb.osmocom.org/trac/.

[3] IEEE Standard Test Access Port and Boundary-Scan Architecture, 1990. IEEE Standard. 1149.1-1990.

[4] IEEE-ISTO 5001 - 2003 the nexus 5001 forum standard for a global embedded processor debug interface. IEEE - Industry Standards and Technology Organization, December 2003.

[5] CWSandbox, 2008. http://www.cwsandbox.org.

[6] AVGERINOS, T., CHA, S. K., HAO, B. L. T., AND BRUMLEY, D. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium* (Feb. 2011), pp. 283–300.

[7] BELLARD, F. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 41–41.

[8] BOJINOV, H., BURSZTEIN, E., AND BONEH, D. Embedded management interfaces: Emerging massive insecurity. In *Blackhat 2009 Technical Briefing / whitepaper* (2009).

[9] CABALLERO, J., YIN, H., LIANG, Z., AND SONG, D. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 317–329.

[10] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008).

[11] CARNA BOTNET. Internet census 2012, port scanning /0 using insecure embedded devices, 2012. http://internetcensus2012.bitbucket.org/paper.html.

[12] CHECKOWAY, S., MCCOY, D., ANDERSON, D., KANTOR, B., SAVAGE, S., KOSCHER, K., CZESKIS, A., ROESNER, F., AND KOHNO, T. Comprehensive Experimental Analysis of Automotive Attack Surfaces. In *Proceedings of the USENIX Security Symposium* (San Francisco, CA, August 2011).

[13] CHING, P. C., CHENG, Y., AND KO, M. H. An in-circuit emulator for TMS320C25. *IEEE Transactions on Education 37*, 1 (1994), 51–56.

[14] CHIPOUNOV, V., AND CANDEA, G. Reverse Engineering of Binary Device Drivers with RevNIC. In *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), Paris France, April 2010* (Paris, France, 2010).

[15] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst. 30*, 1 (Feb. 2012), 2:1–2:49.

[16] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2* (Berkeley, CA, USA, 2005), NSDI'05, USENIX Association, pp. 273–286.

[17] COZZIE, A., STRATTON, F., XUE, H., AND KING, S. T. Digging for data structures. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 255–266.

[18] CRISTIAN, F. Exception handling and software fault tolerance. *IEEE Transactions on Computers C-31*, 6 (1982), 531–540.

[19] CUI, A., COSTELLO, M., AND STOLFO, S. J. When firmware modifications attack: A case study of embedded exploitation. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013* (2013), The Internet Society.

[20] CUI, A., AND STOLFO, S. J. Defending embedded systems with software symbiotes. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection* (Berlin, Heidelberg, 2011), RAID'11, Springer-Verlag, pp. 358–377.

[21] CUI, W., PEINADO, M., CHEN, K., WANG, H. J., AND IRUN-BRIZ, L. Tupni: automatic reverse engineering of input formats. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), ACM, pp. 391–402.

[22] DAVIDSON, D., MOENCH, B., JHA, S., AND RISTENPART, T. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the USENIX Security Symposium* (Washington, DC, August 2013).

[23] DELUGRÉ, G. Closer to metal: Reverse engineering the broadcom netextreme's firmware. HACK.LU 2010.

[24] EGELE, M., SCHOLTE, T., KIRDA, E., AND KRUEGEL, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv. 44*, 2 (Mar. 2008), 6:1–6:42.

[25] FALLIERE, N., MURCHU, L. O., AND CHIEN, E. W32.Stuxnet Dossier, 2011.

[26] FREESCALE SEMICONDUCTOR, INC. MC1322x Simple Media Access Controller Demonstration Applications User's Guide, 9 2011. Rev. 1.3.

[27] FREESCALE SEMICONDUCTOR, INC. MC1322x Simple Media Access Controller (SMAC) Reference Manual, 09 2011. Rev. 1.7.

[28] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated Whitebox Fuzz Testing. In *Network Distributed Security Symposium (NDSS)* (2008), Internet Society.

[29] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. SAGE: whitebox fuzzing for security testing. *Communications of The ACM* (2012), 40–44.

[30] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of USENIX Security'13* (Washington, DC, August 2013), USENIX.

[31] HAN, Y., LIU, S., SU, X., AND HU, Z. A dynamic analysis system for Cisco IO based on virtualization. In *Multimedia Information Networking and Security (MINES), 2011 Third International Conference on* (2011), pp. 330–332.

[32] IEEE COMPUTER SOCIETY. *IEEE 802.15.4, Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate*

Wireless Personal Area Networks (WPANs), June 2006. ISBN 0-7381-4996-9.

[33] KANG, M. G., MCCAMANT, S., POOSANKAM, P., AND SONG, D. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium* (San Diego, CA, Feb. 2011).

[34] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode* (New York, NY, USA, 2007), WORM '07, ACM, pp. 46–53.

[35] KAO, C.-F., HUANG, I.-J., AND CHEN, H.-M. Hardware-software approaches to in-circuit emulation for embedded processors. *Design Test of Computers, IEEE 25*, 5 (2008), 462–477.

[36] KIRCHNER, A. Data Leak Detection in Smartphone Applications. Master thesis, Vienna University of Technology.

[37] KUZNETSOV, V., CHIPOUNOV, V., AND CANDEA, G. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 12–12.

[38] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 193–204.

[39] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), IEEE, pp. 75–86.

[40] LEE, Y.-H., SONG, Y. W., GIRME, R., ZAVERI, S., AND CHEN, Y. Replay debugging for multi-threaded embedded software. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on* (2010), pp. 15–22.

[41] LI, L., AND WANG, C. Dynamic analysis and debugging of binary code for security applications. In *4th International Conference on Runtime Verification (RV) 2013, Rennes, France, September 24-27, 2013. Proceedings* (2013), vol. 8174 of *Lecture Notes in Computer Science*, Springer, pp. 403–423.

[42] MELEAR, C. Emulation techniques for microcontrollers. In *Wescon/97. Conference Proceedings* (1997), pp. 532–541.

[43] MONTENEGRO, G., KUSHALNAGAR, N., HUI, J., AND CULLER, D. Transmission of IPv6 packets over IEEE 802.15.4 networks (RFC 4944). Tech. rep., IETF, September 2007. http://www.ietf.org/rfc/rfc4944.txt.

[44] MULLINER, C., GOLDE, N., AND SEIFERT, J.-P. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale. In *Proceedings of the 20th USENIX Security Symposium* (San Francisco, CA, USA, August 2011).

[45] NOHL, K., EVANS, D., STARBUG, S., AND PLÖTZ, H. Reverse-engineering a cryptographic RFID tag. In *Proceedings of the 17th conference on Security symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 185–193.

[46] PEREZ, Y.-A., AND DUFLOT, L. Can you still trust your network card? CanSecWest 2010.

[47] REDWIRE LLC. Econotag: MC13224V development board w/ on-board debugging. http://www.redwirellc.com/store/node/1.

[48] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. SymDrive: testing drivers without devices. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 279–292.

[49] SCHLICH, B. Model checking of software for microcontrollers. *ACM Trans. Embed. Comput. Syst. 9*, 4 (Apr. 2010), 36:1–36:27.

[50] SCHMITT, P. H., AND WEISS, B. Inferring invariants by symbolic execution. In *Proceedings, 4th International Verification Workshop (VERIFY'07)* (2007), B. Beckert, Ed., vol. 259 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 195–210.

[51] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 317–331.

[52] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011* (2011).

[53] SONG, D., BRUMLEY, D., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. Bitblaze: A new approach to computer security via binary analysis. In *In Proceedings of the 4th International Conference on Information Systems Security* (2008).

[54] TRIULZI, A. A SSH server in your NIC. PacSec 2008.

[55] WANG, T., WEI, T., GU, G., AND ZOU, W. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy* (2010), pp. 497–512.

[56] WEINMANN, R.-P. Baseband attacks: remote exploitation of memory corruptions in cellular protocol stacks. In *Proceedings of the 6th USENIX conference on Offensive Technologies* (Berkeley, CA, USA, 2012), WOOT'12, USENIX Association, pp. 2–2.

[57] WELTE, H. Anatomy of Contemporary GSM Cellphone Hardware.

[58] WILLIAMS, M. ARMV8 debug and trace architectures. In *System, Software, SoC and Silicon Debug Conference (S4D), 2012* (2012), pp. 1–6.

[59] XU, M., BODIK, R., AND HILL, M. D. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture* (New York, NY, USA, 2003), ISCA '03, ACM, pp. 122–135.

[60] ZADDACH, J., KURMUS, A., BALZAROTTI, D., BLASS, E. O., FRANCILLON, A., GOODSPEED, T., GUPTA, M., AND KOLTSIDAS, I. Implementation and implications of a stealth hard-drive backdoor. In *ACSAC 2013, 29th Annual Computer Security Applications Conference, December 9-13, 2013, New Orleans, Louisiana, USA* (New orleans, UNITED STATES, 12 2013).