Towards system-wide security analysis of embedded systems

Nassim Corteggiani

THESE DE DOCTORAT DE
SORBONNE UNIVERSITE
préparée à EURECOM

École doctorale EDITE de Paris n° ED130
Spécialité: «Informatique, Télécommunications et Électronique»

Sujet de la thèse:

Towards System-Wide Security Analysis of Embedded Systems

Thèse présentée et soutenue à Biot, le 31/07/2020, par

Nassim Corteggiani

Président	Prof. Marc Dacier	EURECOM
Rapporteurs	Prof. Jean-Louis Lanet	INRIA
	Prof. Konrad Rieck	TU Braunschweig
Examinateurs	Dr. Renaud Pacalet	Télécom Paristech
	Dr. Marie-Lise Flottes	Univ. Montpellier / CNRS
	Stéphane Di Vito	Maxim Integrated
	Nicusor Penisoara	NXP Semiconductors
Directeur de thèse	Prof. Aurélien Francillon	EURECOM

Preface

Even if only my name is written on the cover of this Ph.D. thesis, many people around me deserved credits for its completion. Therefore, I would like to present my grateful to people who have helped for the preparation of this thesis.

First and foremost a special mention to my supervisor, Aurelien Francillon, who has always been helpful, of good advice, and always present for discussions and last-minute reviews.

I would like to thank my collaborators and co-authors. In particular, I would like to thank Giovanni Camurati. Designing and developing the concepts behind this thesis was a huge undertaking that was successful thanks to his precious collaboration.

Then, I would like to thank my colleagues at EURECOM and at Maxim Integrated, for their feedbacks, discussions and cooperation.

I would like to thank the committee members for their time, support, and guidance during the proofreading of this manuscript. Especially, Prof. Jean-Louis Lanet, Prof. Marc Dacier, and Dr. Renaud Pacalet for their extensive feedback on an earlier draft of this manuscript.

Last but not the least, I would like to thank my family for supporting me spiritually throughout the writing of this thesis and my life in general.

This work was supported, in part, by Maxim Integrated and EURECOM. Any opinions, findings, recommendations or conclusions expressed herein are my own and do not reflect the position or views of any instutition or company.

Résumé

Les systèmes embarqués sont de plus en plus déployés, comme par exemple les objets connectés ou les systèmes de contrôle critique. Leur sécurité devient une préoccupation importante soit parce qu'ils contrôlent des systèmes sensibles ou parce qu'ils peuvent être exploités pour mettre en oeuvre des attaques à grande échelle.

L'une des spécificités majeures des systèmes embarqués reste les interactions fréquentes entre le micro-logiciel et les périphériques matériels qui font l'interface avec le monde extérieur. Ces interactions sont souvent la source de défauts de conception aussi appelés bugs. Une manière commune de tester ces systèmes est l'analyse dynamique. Cependant, les approches existantes se concentrent généralement sur les logiciels dont les sources ne sont pas disponibles ou testent les composants séparemment les uns des autres comme le code binaire, le code source écrit en C, ou les périphériques matériels. Atteindre l'analyse à l'échelle du système est nécessaire pour tester ces systèmes méticuleusement. Les principaux défis dans ce domaine sont la limitation des performances, les différences sémantiques et le niveau de contrôle/visibilité sur les périphériques matériels.

Dans cette thèse, nous nous attaquons à ces trois défis tout en considérant le point de vue du concepteur. Pour commencer, la thèse offre une discussion générale sur l'analyse de système sur puce à l'échelle du système où nous pointons les défis et soulignons les directions de recherche. Pour palier la limitation des performances lors des interactions avec les périphériques (i.e., test avec le matériel dans la boucle), nous proposons Steroids, une sonde USB3 haute performance. Ensuite, nous avons conçu et développé Inception, une méthode complète pour tester le code-source des micro-logiciels à l'échelle du système. Inception supporte différent niveaux de sémantique (e.g., assembleur et langage C) qui sont souvent combinés lors de la programmation de micro-logiciel. Troisièmement, nous proposons HardSnap une solution pour générer des instantanés de l'ensemble du système testé, incluant aussi bien l'état du matériel que celui du logiciel.

Abstract

Connected embedded systems are increasingly widely deployed, for example, in IoT devices or critical control systems. Their security is becoming a serious concern, either because they control some sensitive system or because they can be massively exploited to mount large scale attacks.

One of the specificities of embedded systems is the high interactions between the firmware and the hardware peripherals that generally interface them with the real world. These interactions are often the source of critical bugs. One common way of testing such systems is dynamic analysis. However, current approaches generally focus on closed-source firmware and rely on testing components separately such as binary code, C-based code, or hardware peripherals. Achieving system-level testing is necessary to thoroughly test these systems. Major challenges in this topic include performance limitations, semantics differences, and limited control/visibility on hardware peripherals.

In this thesis, we tackle these three main challenges for system-level dynamic analysis of embedded systems while taking the point of view of a designer. To begin with, this thesis offers a general discussion on achieving a system-wide analysis of System-on-Chip (SoC) where we point out challenges and highlight research directions. To overcome performance limitations when interacting with peripherals (i.e., hardware-in-the-loop testing), we propose Steroids, a USB3-based high-performance low-latency system probe. Second, we designed and developed Inception, a complete solution for testing system-wide firmware programs source-code. Inception supports different semantics levels (e.g., assembly and C), which are often combined when writing the firmware program. Third, we propose a solution for snapshotting the entire system under test, including both hardware and software *state*. We implement this solution in HardSnap, a system that enables system restoration at a precise point for testing multiple execution paths concurrently while preserving analysis consistency.

Contents

List of Figures

Glossary

SoC System-on-Chip

PoC Proof of Concept

IP Intellectual Property

I/O Input/Output

SPI Serial Peripheral Interface

UART Universal Asynchronous Receiver-Transmitter

I2C Inter-Integrated Circuit

IC Integrated Circuit

USB Universal Serial Bus

DMA Direct Memory Access

MMIO Memory Mapped Input/Output

PMIO Port Mapped Input/Output

DSE Dynamic Symbolic Execution

BSP Board Support Package

EGT Execution-Generated Testing

ISA Instruction Set Architecture

API Application Programming Interface

RTL Register-Transfer Level

ROM Read Only Memory

FPGA Field Programmable Gate Arrays

AES Advanced Encryption Standard

SHA Secure Hash Algorithms

CPU Central Processing Unit

IoT Internet-of-Things

ICE In-Circuit Emulator

OCD On-Chip Debugger

JTAG Joint Test Action Group

PCI-E Peripheral Component Interconnect Express

DCC Debug Communication Channel

IDE Integrated Environment Development

AHB-AP AMBA High-performance Bus Access Port

IR Intermediate Representation

ELF Executable and Linkable Format

ABI Application Binary Interface

BSS Block Started by Symbol

ICMP Internet Control Message Protocol

HTTP Hypertext Transfer Protocol

OTP One Time Programmable

SRAM Static Random-Access Memory

SDK Software Development Kit

VM Virtual Machine

HW Hardware

SW Software

HDL Hardware Description Language

SSE Selective Symbolic Execution

AXI Advanced eXtensible Interface

PIC Programmable Interrupt Controller

Chapter 1

Introduction

Computer systems are ever more present in all aspects of our modern life. Some compelling examples of such products are cars, personal computers, home appliances, or smartphones. These systems range from the well-known general-purpose computing devices (e.g., personal computer, smartphone, or server) to purpose-built computing devices, so called embedded system.

Historically, embedded systems have been closely linked to the commodity they embedded. They are designed for specific purposes. For instance, the micro-controller in a washing machine that tunes motors speed up. Embedded systems are also an ubiquitous blocks of general-purpose-computer. They handle interactions with mechanical (e.g., hard drive) or electronic systems (e.g., optical drive, or image sensor).

However, embedded systems are now becoming ever more complex, connected and often attached to the Internet network. This forms the so called "Internet of Things" where today's objects are connected to online services through the Internet network. From connected cars to home appliances, and wearables, there is a considerable number of computer systems that emerge in our lifes. It is difficult to precisely quantify the number of deployed connected devices. However, a reference could be the forecast dataset published by Gartner which tracks connected devices in the following markets: healthcare, smart buildings, smart cities, retail, agriculture, utilities, transportation, manufacturing, automotive. According to Gartner [48], 4.81 billions of connected devices are on the market in august 2019. In addition, it forecasts 5.8 billions of endpoints in 2020 that represents a 21% increase from 2019.

The semiconductor market is growing with tremendous volumes making it attractive but also extremely competitive with short time-to-market. As repeatedly proved by the recent litterature, this strain generally lets little room for testing. In addition, the growing complexity of embedded systems makes security analysis challenging. These systems are very often based on micro-controllers running firmware programs that are often written in programming languages that do not protect from memory corruptions (e.g., C/C++, assembly).

The consequence of a vulnerability can be disastrous. First, embedded systems often control a physical or electrical systems in the real world. An unexpected behavior may cause human harm (e.g., industrial robotic arm) or impact the security of real-world infrastructure (e.g., traffic light system). Second, they may collect personal information issued from sensors or interactions with other connected devices, and therefore may reveal individual private life such as habits, location, interests or even credit card numbers. Third, vulnerable connected devices that interact with each other enabling

large scale attacks.

In addition, fixing a vulnerability on embedded systems is not always possible. It is rare to find devices with an automated remote update mechanism enabling security updates. Furthermore, some parts of these systems are hardwired, and it requires hardware replacement to fix bugs. This is the case for any silicon-based hardware components and firmware stored on mask ROM. Fixing these parts requires costly re-fabrication as it involves a new wafer mask set that usually costs more than 1×10^5 \$. As a consequence, the economical impact for the manufacturer or product owner can be dramatic and may affect its brand image. One compelling example of such issue is the recent vulnerability discovered in the Nvidia Tegra chips embedded in the Nintendo Switch [80]. This security issue is a memory corruption present in the mask ROM bootloader. Nintendo could only correct this issue in new versions of the console equiped with an updated system on chip. And had no choice other than leaving the vulnerability exploitable in all previously manufactured devices.

1.1 Context

In this thesis, we take the point of view of a chip manufacturer who is interested in testing chips before manufacturing. For the sake of clarity, we provide a description of this process. Generally, the design of the core is provided by a third-party (e.g., ARM Architecture). Then, this design is extended with custom and re-utilizable hardware blocks also called IP blocks. These blocks form the hardware peripherals that offer a gate to the real world. The level of customization for these blocks can be important especially for Application-Specific Integrated Circuit (ASIC). In this case, IP blocks are customized to fit specific needs. During the chip development cycle, hardware and software are built concurrently to fit short time-to-market. For this reason, chip manufacturers generally test firmware programs and hardware peripherals separately on an emulation or simulation platform. However, testing components separately is often inefficient for detecting bugs that are due to the interactions of different components such as hardware and firmware. For this reason, performing system-wide testing is crucial but challenging due to the short time-to-market and the lack of dedicated tools.

For all these reasons, there is a need for security testing tools to thoroughly test embedded systems software in pre-production.

1.2 Problem Statement

The growing complexity of embedded systems has been possible with the deployment of chip that mixes firmware programs and hardware peripherals. This latter offer either an interface to the real world, accelerated computation or custom functionalities. By analogy, hardware peripherals are what software libraries are for desktop applications. They may have intricate semantics and complex interactions with the firmware. Furthermore, these interactions may be the source of critical bugs that may have dramatic impacts in the real world.

In computer science, previous work highlighted promising perspectives for testing complex systems using dynamic analysis techniques ([39], [93], [98], [53], [101], [87], [58], [31], [90]). Research in this field is hindered by a lack of methods for applying these approaches on embedded systems and even more for source-based analysis. Furthermore, most of those solutions address binary only testing. In particular, they often rely on closed-source only testing and limit the analysis to some components of the systems.

In this thesis, we try to answer this question: How much effort do we need to apply system-wide dynamic analysis to embedded systems while considering the point of view of a chip manifacturer?

1.3 Thesis Outline

In Chapter 2 "State-of-the-Art", an overview of the state of the art in the field of the source-based security analysis of embedded systems is presented; in Chapter 3 "SoC Security Evaluation: Reflections on Methodology and Tooling" we present a reflection on security analysis of embedded systems, and we point out the main challenges that we seek to tackle in this thesis; in Chapter 4 "Steroids", a fast probe for optimizing existing dynamic security analysis approaches; the work presented in Chapter 5 "Inception: System-Wide Security Testing of Real-World Embedded Systems Software" illustrates a novel approach to support real-world firmware program analysis where in practice different semantics levels are mixed (e.g., inline assembly, C/C++ and binary code); in Chapter 6 "HardSnap: Leveraging Hardware Snapshotting for Embedded Systems Security Testing", we present a technique for snapshotting both firmware and hardware peripherals to enable advanced dynamic analysis techniques without inconsistencies and with performance in mind.

The thesis ends in Chapter 7 with the conclusions and future perspectives.

Chapter 2

Background

This chapter describes the background relevant to this thesis. To begin with, we detail the threats and the risks that computing systems and more specifically embedded systems face. Then, we define and explain what are embedded systems and how they differ from the traditional desktop computer. Thereafter, we discuss computer testing methods in general and what challenges to overcome when applying these methods to embedded systems. Finally, we present the state of the art on dynamic embedded systems source-code analysis.

2.1 Threats and Risks

Nowadays, computer systems are ever more present in all aspects of our modern life. The increasing complexity and connectivity make them more exposed to attacks. After 40 years of history, the main fundamental root cause of attacks remains memory corruption in software programs. The recent MITRE [67] study shows up memory corruptions among the 3 most widespread and critical weaknesses in software. According to [84], 50% of discovered security bugs in Google Chrome, a popular web-browser written in C/C++, are memory corruptions (i.e., use-after-free, buffer overflow, uninitialized memory). Even if memory-safe languages are available, memory-unsafe languages such as C remains very popular. For instance, the C-language is commonly used for programming software such as Operating Systems (e.g., Linux, Windows, Mac OS, or Android) or payment systems. Exploiting memory corruptions may lead to private information theft, physical infrastructure outage (e.g., bank, health emergency systems), real-world damage (e.g., traffic lights, robotic arms).

Embedded systems are no exception ([10],[11], [12], [13]). Firmware programs are often written with low-level programming languages (e.g., C, C++, assembly) to handle low-level interactions with the hardware. These languages lack memory safety mechanisms (e.g., types checking) and therefore, they frequently lead to memory corruptions. Although memory safe languages have been used in the past for safety-critical systems ([92]) and have been gaining popularity recently ([57, 43, 20, 23]). An overwhelming part of the development of embedded systems is done as a mix of assembly and C.

The consequences of such vulnerabilities can be disastrous. Since embedded systems may drive mechanical systems in the real world, they may cause human harm or damage physical infrastructures. For instance, the Stuxnet [60] worm caused substantial damage to the nuclear program of Iran. Vulnerability may also lead to considerable economic losses for the

chip manufacturer, especially when fixing the vulnerability requires costly re-fabrication [80].

There is therefore an important need for security testing tools to automatically detect memory corruptions on firmware programs before production.

2.2 Embedded Systems

Embedded systems are now as ubiquitous as the traditional personal computer. The high spread of these systems is tightly coupled with the deployment of micro-controllers. These computer systems reduce the size, cost, and power-consumption of design by embedding all the necessary components to fulfill specific needs inside a single chip. The latter is also referred to as a System-on-Chip. This approach differs from the traditional microprocessor that depends on other external components to run (e.g., memory, motherboard, Graphics Processing Unit) and is designed for genereral purpose.

Embedded systems are heterogeneous systems with a wide variety of architecture where the following components are generally customized: the computer architecture bit widths, the presence of a data/instruction cache, the Instruction Set Architecture, the peripherals, the memory protection mechanisms, the memory size/type, the presence of a co-processor. In the following, we present the two common key components of chip that are mixed together to achieve specific tasks.

2.2.1 Firmware

Embedded systems are often software-driven and combine hardware peripherals and firmware programs. They are often purpose-built computers and therefore involve a high level of customization for both software and hardware. These systems may run a variety of software that can be divided into three categories. We follow the classification from [70] where authors identify three main classes of firmware in embedded devices.

- **General-Purpose Operating Systems.** These operating systems are designed for versatility, and they can, therefore, address a large variety of applications. Some prominent examples of general-purpose OSs are Linux-based OSs such as Debian [1], Fedora [2], and Arch-Linux [3]. Linux is well-spread because of its free-cost and customization capability. However, such complex OSs generally requires important resources such as memory, CPU time and power-consumption.

Therefore, they are reserved to a powerful category of embedded devices. Such operating systems introduce an abstraction layer making running applications independent from the underlying hardware, and they offer a testing environment relatively similar to desktop programs.

- **Embedded Systems Operating Systems.** To address time/space constraints, custom operating systems have been developed (e.g. VxWorks [91], QNX [24], FreeRTOS [21]). Generally, these systems provide an isolation layer between user-space and kernel space thanks to a Memory Protection Unit (MPU) or a logical separation (i.e., context switching). Commonly, embedded devices have the kernel code and the applications code together in a monolithic block [94].

- **Bare-metal Firmware.** It is also common to find firmware programs running in bare-metal without any Operating System. This approach fits well when the firmware programs have strong time constraints or space constraints. It is generally the case for boot-loader that aims at high performance and low-memory footprint.

2.2.2 Peripherals

To meet specific requirements, system-on-chip blend software and hardware peripherals together. Such peripherals are predesigned digital circuits, often referred to as Intellectual Property (IP) blocks, acquired from internal sources or from third parties. Generally, a peripheral can be intern to the chip or extern, in this case, the communication with an external peripheral passes by a bus generally driven by an internal peripheral (e.g., SPI, I2C, UART). There is a large number of IP, however, they are generally designed for one of these three purposes.

- **Hardware Accelerator.** For performance and security purposes, algorithms may be implemented using digital circuits. This solution offers higher performance than full-software stack and it enables the algorithm to be separated from the rest of the machine. This is generally the case for cryptographic blocks to avoid any leakage to the software.

- **Hardware Input/Output Peripherals.** Peripherals are often a gate to the real world where they sample information from sensors or interact with actuators. Some compelling examples are temperature sensors, gyroscopes, proximity sensors, and buttons. I/O peripherals

may also offer a communication interface to drive external peripherals (e.g., SPI, I2C, UART) or transfer data to another device (e.g., USB, Bluetooth, or WiFi).

- **Hardware Assisted Features.** Finally, peripherals can be classified as hardware support to software to optimize repeated operations (e.g., Direct Memory Access), or to enable event-based programming (e.g., interrupt controller).

2.2.3 Hardware/Software Interactions

Generally, firmware programs are interrupt-driven and interact with the underlying hardware through different mechanisms. These interactions are frequent, numerous and somewhat complex. In fact, peripherals may affect the system's memory (i.e., change the data-flow) and interrupt the firmware execution (i.e., change the control-flow). Therefore, testing firmware programs involves a clear understanding and consideration for these interactions. In the following, we describe such mechanisms.

- **Interrupt Signal.** Firmware programs are event-driven software where execution is frequently interrupted by interrupt signals emitted by peripherals. These asynchronous signals notify the software that a specific task is done. This concept is largely present in embedded systems to reduce the power consumption (an inefficient alternative would be polling) and to fit relatively strong time constraints for some specific tasks.

- **Memory Mapped Input/Output (MMIO).** A common way for the firmware to drive peripherals is to expose some registers to the firmware address space. This mapped memory enables the firmware to directly interact with internal peripherals.

- **Port Mapped Input/Output (PMIO).** An alternative to the MMIO is the use of dedicated instructions enabling data transfer to peripherals registers through the processor.

- **Direct Memory Access (DMA).** One specific case of interaction is the Direct Memory Access that is extremely prevalent in embedded systems. It enables firmware/hardware to initiate a transfer of memory chunks between RAM and peripherals' memory or the opposite. It is often implemented by peripherals that require large data transfer such as Ethernet, or WiFi cards.

2.3 Static and Dynamic Analysis

Over the years, many security analysis techniques have been proposed. These approaches can be divided into static and dynamic analyses.

Static analysis examines the computer program code without actually executing it. It is possible to achieve a sound analysis. Sound means that giving an expression the analyzer is able to verify that it is always true. In other words, if the analysis claims that an expression X is true then it is actually true. However, in practice such tools often approximate the tested program environment, and are therefore often not complete in the sense that they may report false positives, i.e., a violation that would never happen in reality. In fact, completeness is such that if an expression X is true, the analysis claims that X is true. However, static analysis is often used as a context insensitive method, and is therefore often imprecise. In particular, the program environment is generally modeled using an over-approximation that significantly increases the false positive rate. Some values may not be possible when executing the program under test.

On the contrary, the dynamic analysis methods execute the program concretely in order to limit the execution to realistic values (i.e., context sentitive approach). In fact, the tested program often interacts with its real environment during the analysis. One of the most widely-deployed dynamic analysis technique is certainly fuzzing. The term ‘fuzz‘ was introduced in 1990 by Milleret et al. [66]. Fuzzing runs a test program in a loop while feeding this program with different grammatically or syntactically malformed inputs at each iteration. Intuitively, the efficiency of fuzzing relies on the quality of the test inputs. These inputs exercise part of the code and therefore affect the code-coverage and the accuracy of the vulnerability detection. Generating relevant test cases is challenging. One hindering factor is the presence of complex control-flow (e.g., hashing functions on program inputs) that requires a program inspection to explore the state space of the program systematically. This systematic exploration of the program under test is achieved by a technique called Dynamic Symbolic Execution (DSE) [29] that basically reasons about the program execution to determine which inputs cause which part of the program to be executed. For the sake of clarity, we detail this technique later on. Before going further, we discuss the analysis context and how it affects results.

2.4 Binary and Source Analysis

The context under which the security analysis is performed varies a lot. Many approaches base their analysis on binary code ([98], [69], [33], [22], [90], [87], [53], [58], [101]). We emphasize the fact that in the wild many firmware programs are only available in binary form. This approach is generally performed by third parties (e.g., audit, pentesters). However, during the compilation process, most of the source-code semantics, which is crucial for detecting bugs, is gone. This makes binary code analysis more challenging. On the contrary, when the source code is available, the analysis benefits from the availability of high-level semantics (e.g., the types of variables). This semantics information highly simplifies the detection of memory corruptions. There are very few tools for source-based analysis and even less for embedded systems ([39], [93]). Furthermore, the analysis of firmware source-code is not straightforward. In fact, firmware programs are often a mix between high-level programming languages (e.g., C/C++) and low-level programming languages (e.g., inline assembly) to handle interactions with the underlying hardware. In addition, it is common to find binary dependencies (e.g., Board Support Package). Existing firmware source-based analysis approaches either replace unsupported low-level code with handwritten code [39] or stop their analysis when executing low-level code [27].

2.5 Dynamic Symbolic Execution

Symbolic execution has been first introduced by J. C. King et al. [56]. This technique is recently getting more popular due to the recent progress on constraint solving. Symbolic execution automatically discovers which inputs cause which parts of the program to be executed. As the name suggests, this technique replaces concrete input values by symbolic input expressions. As a result, the program execution outputs expressions as a function of these symbolic inputs. These expressions are solved by a constraint solver that generates concrete test cases.

Originally, symbolic execution was a static code examination where the analyzer maps symbolic expressions for each variable [29]. However, this approach has two main limitations. First, the programs may generate expressions that cannot be solved by the constraint solver. Second, it fails at handling interactions with external functions. These two problems are alleviated by dynamic analysis techniques such as Execution-Generated Testing introduced by [28] and followed by [27]. EGT executes the program oper-

ations just like the original program and performs symbolic computation only when one of the operands is symbolic. This reduces considerably the size of the symbolic expressions and enables the computation of non-linear arithmetic that would be difficult or expensive in a symbolic form. Furthermore, EGT supports interactions with external functions by using the concrete form.

In software testing, symbolic execution offers a high code coverage but often suffers from the well-known path explosion. In fact, the number of execution paths grows exponentially with the program size (i.e., conditional branch instructions). This is exacerbated when the firmware environment is abstracted ([39]). In particular, peripheral can be treated as a stateless untrusted function where writes are ignored and reads return unconstrained symbolic values. However, this leads to the explosion of the number of possible paths and consider paths that are not actually feasible with the real peripherals (false positives). Different strategies have been proposed to limit this problem. Among them, Partial-Emulation that we present below.

2.6 Partial-Emulation Testing

Firmware programs are often designed to run in a resources-limited environment where time and size constraints are consequent. Furthermore, low-power micro-controllers generally lack virtualization and memory isolation making fault detection extremely difficult when running the program on the real device [71]. To overcome this problem, recent work proposed emulation ([52], [72], [31]). The firmware program is executed in a virtual machine that is composed of an Instruction Set Architecture (ISA) emulator and behavioral models for peripherals. This virtual-machine offers higher introspection (i.e., visibility and control) on the firmware. However, peripherals may have intricate semantic making a behavioral model difficult to write and error-prone. Furthermore, the interactions between firmware and peripherals are essential for the analysis as vulnerabilities may source from the specific interaction between firmware and hardware. In fact, the interactions between firmware and hardware are numerous and frequent. These interactions affect both firmware and hardware control-flow. A general approach to this problem is partial-emulation. This method has been first introduced by Avatar [98] and followed by [58] [93] [90] and consists in re-hosting the firmware execution in a virtual machine while forwarding interactions to the real device. In particular, when the firmware program accesses memory-mapped addresses, Inputs/Outputs are seamlessly forwarded to the real device through a debugger. This method reduces the complex-

ity of the analysis at the price of completeness. However, in practice, the latency introduced by the I/O forwarding is generally too important and affects significantly the analysis performance.

	FIE	Avatar	Surrogates	Prospect	FirmUSB	Avatar2	Charm	Firm-afl	Firmalice	Firmadyne
	2013	2014	2014	2014	2017	2018	2018	2019		
	[39]	[98]	[58]	[90]	[53]	[69]	[93]	[101]	[87]	[31]
Firmware Type	BM	BM/ESOS	BM/ESOS	GPOS	BM	BM/ESOS	BM	GPOS	GPOS	GPOS
Fast Forwarding	✗	✗	✓	✓	✗	✓	✓	n/a	n/a	n/a
Can use real peripherals	✗	✓	✓	✓	✗	✓	✓	✗	✗	✗
Symbolic Execution	✓	✓	✗	✗	✓	✓	✗	✗	✓	✗
Using source code	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
Inline assembly	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓
Binary code	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓
Context Sensitivity	I	S	S	S	I	S	S	S	I	I
Ensure HW/SW Consistency	n/a	✗	n/a	n/a	n/a	✗	✗	✗	✗	✗
Full Controllability	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
Full Visibility	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
Open-source	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓

Table 2.1: Related work comparison. BM: Barre Metal; ESOS: Embedded Systems Operating System; GPOS: General Purpose Operating System. Context Sensitivity: Insensitive; Sensitive

2.7 Challenges

In the previous part, our literature review points out that modern software testing approaches merely focus on testing desktop applications. Few methods are designed for specifically testing the security of embedded systems and even less for system-wide or source-based analysis. In this thesis, we identify and tackle the three main challenges for applying modern dynamic analysis methods to embedded systems. For the sake of clarity, we present a comparison of existing firmware analysis techniques in Table 2.1. The comparison of previous work highlights fundamental challenges that narrow the dynamic analysis of firmware programs.

Low latency dynamic analysis for embedded systems. Partial-emulation is commonly used for testing firmware programs. This technique runs the firmware to test in a controlled environment such as an emulator. However, we showed that hardware and firmware have complex interactions that are difficult to abstract. For this reason, partial-emulation techniques generally forward I/O to the real peripherals using a remote debugger. However, this communication is generally extremely slow making the analysis impractical.

Firmware source-code semantic differences. Most existing techniques focus on analyzing either binary code or source code but neither considers realistic cases where the source-code contains inline assembly or depends on closed-source binary libraries (e.g., Board Support Package). In fact, as we will show in our study in Chapter 5, real-world programs often mix C/C++ and inline assembly to interact with the underlying hardware. Designing a hybrid analysis technique (i.e., binary and source together) is necessary for achieving a system-wide analysis of real-world programs.

Controlling/observing both firmware/hardware peripherals. Generally, testing embedded systems involves a full-control and full-visibility over the system under test. This enables to inspect the internal state of the system in order to detect design flaw. However, achieving this level of introspection is challenging for embedded systems that mix firmware and peripherals. Existing methods face a trade-off between performance and visibility/control.

2.8 Contributions

In this thesis, we tackle the main challenges research face to apply modern source code security analysis techniques to embedded systems software. In particular, we make the following contributions:

- A reflection on System-on-Chip dynamic analysis where we point out challenges and possible research directions.
- Steroids, a low-latency and high performance USB3-based Debugger. Our solution overcomes the current performance limitation of dynamic analysis techniques where the interactions with hardware pass through a debugger device.
- Inception, a novel framework for system-wide dynamic analysis of firmware programs. This tool is the first dynamic symbolic execution engine that fully supports the security analysis of firmware programs source code. We analyzed different widespread firmware programs and observed that even when the source code is available, it is often a mix of different programming languages having different semantics levels.
- HardSnap a technique for both hardware/software snapshots. This novel method is relevant for many embedded systems dynamic analysis tools that require system-level state manipulation. We implemented this method on top of Inception and demonstrate the performance enhancement during analysis. Furthermore, we identify and explain possible inconsistencies when HardSnap is not enabled.

2.9 Publications

This thesis has led to three publications.

- **Corteggiani Nassim**, Giovanni Camurati, Marius Muench, Sebastian Poeplau and Aurélien Francillon. "SoC Security Evaluation: Reflections on Methodology and Tooling." Invited paper submitted at IEEE Design & Test. Chapter 3.
- **Corteggiani Nassim**, Giovanni Camurati and Aurélien Francillon. "Inception: System-wide security testing of real-world embedded systems software." 27th USENIX Security Symposium (USENIX Security 18). 2018. Chapter 4 and 5.
- **Corteggiani Nassim** and Aurélien Francillon. "HardSnap: Leveraging Hardware Snapshotting for Embedded Systems Security Testing." 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2020. Chapter 6.

Chapter 3

SoC Security Evaluation: Reflections on Methodology and Tooling

The growing complexity of Systems-on-Chip challenges our ability to ensure their correct operation, on which we rely for more and more sensitive activities. Many security vulnerabilities appear in subtle and unexpected ways in the interaction among blocks and across layers, where current verification tools fail at catching them or do not scale. For this reason, security evaluation still heavily relies on manual review. Inspired by the Hack@DAC19 contest, here bellow, described here below, we present our reflections on this topic from a software and system security perspective. We outline an approach that extends the dynamic analysis of firmware to the hardware.

3.1 Introduction

One of the driving factors for the growth of the electronics industry is its pervasiveness in other sectors. Embedded and connected devices are largely present in physical systems, such as cars and industrial plants, where they have a huge impact on safety. In addition, more and more sensitive activities, such as payments and voting, are carried out with digital equipment.

In this context, a major challenge is ensuring the correct operation of an SoC and its software, while satisfying strict requirements in terms of functionality, cost, and time to market. Abstraction and separation of layers, in particular hardware and software, are effective ways to cope with complexity and make the design and verification of such systems possible. However, many security vulnerabilities originate precisely from unexpected interactions between layers and components.

On one side, traditional techniques fail at catching these cross-layer problems, or do not scale to real-world designs. On the other side, identifying novel methodologies is hard because researchers often do not have enough access to all the parts of the system. This is particularly true for proprietary hardware micro-architectures. Hack@DAC is a security contest designed to overcome this problem and stimulate research on the automation of security analysis. The contestants have to find software-controllable hardware vulnerabilities in an open-source design, in which the organizers have injected real-world security bugs.

Based on our experience at Hack@DAC19, in this chapter we present our thoughts on SoC security testing from the point of view of software and system security. We first review the typical goals as well as the constraints of a security analysis (Section 3.2). Then we describe our methodology, as applied to the two rounds of the Hack@DAC19 contest (Section 3.3). Finally, we explore research opportunities to reduce or eliminate manual

aspects in SoC security analysis and to benefit from synergies between the hardware and the software testing communities (Section 3.4). We believe that our background in software and system security gives us an interesting perspective on the problem of hardware/software co-design.

3.2 Background

In this section, we describe the setup and the objectives of the security evaluation before discussing our methodology in the next section.

3.2.1 Security evaluation of SoCs and their firmware

The goal of any security evaluation is to establish a system's conformance to a specification of security properties.[1] In the context of SoCs, both hardware and software play an important role.

Software typically abstracts from the low-level details of the hardware it is running on. However, the validation of software against functional and security specifications needs to violate this abstraction for several reasons. First, an increasing number of security features relies on the interaction with complex hardware mechanisms, which cannot be blindly trusted. Even small hardware bugs may undermine the security of the software layers above. Second, embedded software is often intimately connected with hardware components such as the peripherals, and cannot be easily analyzed without taking this relation into account.

On the hardware side, designers need to take software concerns into account. While the traditional verification and testing flow is mature and guarantees a high level of functional correctness at the hardware level, conventional techniques fail to capture the cross-component, cross-layer interactions that may transform small hardware problems into catastrophic security flaws.

Bugs in the blocks that compose the memory interconnect are a typical scenario. For example, unprivileged code may gain access to an encryption key if a secure register is erroneously mapped to unprotected memory. Similar problems can occur when address ranges overlap, or when a peripheral with access to protected memory can be manipulated [41].

[1] In this chapter, we use the terms *security analysis* and *security evaluation* to refer to the general process of investigating the level of security of a system, independently of the techniques and specifications used (e.g., formal verification, simulation, FGPA emulation, validation, dynamic analysis).

In this chapter, we focus on a methodology that is targeted at finding precisely those bugs that arise from cross-layer interplay between software and hardware.

3.2.2 Analysis context

Security analysis may take place in different scenarios. Literature on embedded software security tends to focus on the black-box case, in which only the binary firmware (or the source at best) and the silicon device are available, but the internals of the hardware are unknown. On the other hand, chip manufacturers have access to the RTL code [2] of the hardware design (provided that they do not use black-box IPs) but may not know the software that will run on it. Vendors providing integrated solutions have access to both firmware source code and hardware internals.

Likewise, the security specification of the system under test may or may not be available. Its level of detail can vary greatly, and security properties may be expressed in a formal or informal way. Ideally, system designers have access to models of the system and its required properties at a high level of abstraction. Finally, the goal of the analysis can vary from finding individual violations of security properties (e.g., in an adversarial context where a single vulnerability is sufficient to compromise a system) to the quest of full validation, i.e., proving the absence of violations under all circumstances.

In this chapter, we take the point of view of third party security analysts who have access to an RTL description of the hardware, C source code developed for it, and informal specifications of the expected security properties. The goal of the analysis is to find as many security issues as possible in a limited amount of time, but the specification is not precise enough (and the time is not sufficient) to allow for formal verification. Hack@DAC provides an example of this setting for public research, as we explain in the following.

3.2.3 SoC and firmware at Hack@DAC19

Hack@DAC is a security contest that simulates the task of security analysts at a chip vendor under pressure to ship the product. Participants have to find security vulnerabilities in an SoC, with a focus on bugs that can be exploited from software, called HardFails [41]. The 2019 edition was based on Ariane, a 64-bit RISC-V processor that is able to boot Linux. The design

[2] Register Transfer Level (RTL) is a method for describing the behaviour of a microelectronic circuit.

had been extended with additional features, such as a password-protected JTAG debug port, an AES cryptographic engine, a secure ROM with secure registers, a peripheral to select encryption keys, and access control for mapped memory. The contestants had access to the RTL Verilog code, a toolchain and a testbench to simulate the execution of C programs, and a brief natural-language description of the security features of the system. For the finals, the firmware with the APIs to access the peripherals in the intended way was also available. In addition to any vulnerabilities that may already have been present in the system, the organizers injected several real-world vulnerabilities "donated" by hardware vendors.

3.3 Security evaluation methodology

In the following, we give a detailed description of the methodology we followed to find bugs on the Hack@DAC platform.

3.3.1 Requirements on tooling

The preliminary step of any security analysis consists in choosing the right tools for the investigation. In the given context, we formulated the following requirements:

1. Tools must support RTL, C and hand-written assembly code (e.g., Ariane SoC bootloader). The system under test is made of different blocks of hardware and software that interact with each other. These interactions are sometimes complex and may lead to security issues that expose the entire SoC. The trigger conditions associated to these kinds of bugs require putting multiple components of the system in a specific state, potentially involving firmware and various hardware blocks. Therefore, analysis tools need to support both RTL analysis and firmware execution.

2. We need an easy way to express security properties. Time constraints and the vague specification forbid elaborate formulations of expected behavior in a formal language.

3. Time is of the essence, so that we need to find bugs quickly. Given a security property and the platform under test, the tool should determine in a relatively short amount of time and with high confidence whether the property holds.

Tool	C&RTL support	Sufficiently expressive	Analysis Complexity	Set-Up Time
FPGA	Y	Y	Low	Long
Verilator	Y	Y	Low	Short
Model Checking	Y	Y	High	Long
Theorem Prover	N	Y	High	Very Long
KLEE	N	Y	High	Short

Table 3.1: Non-exhaustive comparison of state-of-the-art analysis techniques for SoC testing.

4. For the same reason, we need tools that are fast to set up. It is very difficult to estimate the efficiency of any single tool on a given design and security specification. Therefore, we chose to avoid tools that require a significant set-up time and rather allow for combinations of several tools.

3.3.2 Available tools

With these goals in mind, we compared available techniques; see Table 3.1. It is important to remember that, due to time pressure, our goal is to show the *presence* of security violations, not to prove their *absence*. The latter is much more difficult because it needs to exhaustively reason about all possible states of the system. Discovering individual vulnerabilities, in contrast, is less time-consuming as we only need to find a single execution state in which a security property is violated. The intuition is that by iterating the process we approach a state that is indistinguishable from a fully validated system.

We quickly excluded theorem provers and model checking for several reasons. Both techniques generally assume deep knowledge of the system, require significant time to set up and, most importantly, they make it hard to express correctness properties over complex states, such as interactions between multiple hardware blocks. Furthermore, model checking is affected by the state-explosion problem, which we consider a severe obstacle when reasoning about a system as complex as an SoC in limited time.

We like to use software as a means of describing behavior at a high level of abstraction, which makes it easier to test security properties and push the system in a specific state: instead of considering low-level interactions in the electronic circuits, we focus on communication between logic

blocks (e.g., AES engine, DMA, access control system) and use custom firmware to find cases where specific security properties fail. We settled on a software-centric workflow built around quick experiments with component interaction, driven by software.

3.3.3 Methodology

Our methodology for analyzing the security of an SoC comprises three main steps:

1. *Security properties.* A common challenge for all analysis techniques is the need to describe the desired security properties. Ideally, testers receive a specification containing a precise description of the security properties that the implementation should respect. But in reality, the specification is usually an ambiguous document mixing functionality and security requirements. Therefore, testers typically need to employ intuition, experience and a good amount of skepticism in order to develop hypotheses of potential failures. This step requires studying the literature, documentation, and the implementation (RTL and firmware) to identify suspicious control or data flows that could result in bugs. We complement this manual work with static analysis (Verilator sanity checks and symbolic execution of the firmware with predefined sanity checks). However, while firmware analysis tools can efficiently find memory corruption in firmware, they encounter limitations when searching bugs related to digital hardware components.

2. *Writing a Proof of Concept (PoC).* Once we have an hypothesis on a potential violation of a security property, we test this hypothesis in a two step process. First, a block of C code drives the hardware to reach the specific state that is assumed to be vulnerable. At this level of abstraction, it is easy to drive all hardware components as needed. Then our program checks whether the security property in question has indeed been violated. Again, the higher level of abstraction in software tests lets us conveniently express cross-component constraints.

3. *Running the PoC on a dynamic analysis platform.* To evaluate our PoC against the system under test, we use a cycle-accurate simulation of the RTL code with the goal of executing C code on the Ariane SoC.

Figure 3.1 illustrates the methodology. While this methodology is by no means a way of guaranteeing the absence of bugs, we believe that it mirrors the reality in many cases where full verification is too costly to make sense economically.

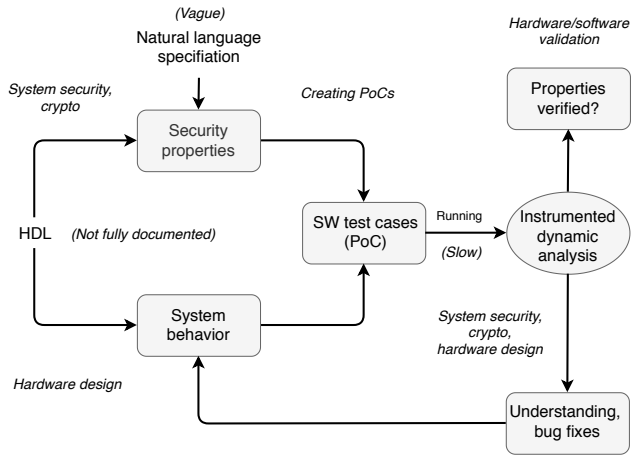


Figure 3.1: Overview of the security evaluation methodology based on dynamic analysis. Manual work and human expertise from different fields are marked in italic.

3.3.4 Results from Hack@DAC19

We applied the presented methodology at the Hack@DAC19 competition and provide insights about the awarded scores and achieved results in the following. The competition had two different phases: the qualification (10 weeks) and the final (3 days). The design for the qualification has 44975 lines of SystemVerilog code, while the design for the final has 47282 lines of code. We participated as the academic team NOPS and led the qualification phase for the first 5 weeks, finishing second (240 points) after the team Hackin' Aggies (300 points), and before the other 13 teams (less than 200 points). During the finals, we achieved the first place in the academic bracket with 330 points, just after the academic/industrial team Hackin' Aggies (465 points), and before the other 11 teams (less than 290 points).

The 2018 edition presented similar classes of bugs on a different platform. To the best of our knowledge, we found all those that were based on similar problems. A detailed list of the bugs can be found in [41], together with a description of the techniques used to find them. The authors observe that classic techniques often fail to capture cross-module cross-layer bugs, and that formal properties are often hard if not impossible to write, even when already knowing the bug. On the contrary, our test vectors in software were often straightforward to write, as this is a good layer to stress several blocks at the same time and to bring the system in the desired overall state. In total, we found 29 bugs not listed in [41] out of 32 reported bugs.

Not requiring neither complex tools nor a detailed knowledge of the hardware design, our methodology was well suited for a fast-paced competition, which mimics strict deadlines typically encountered in industrial settings. Our focus on system/software aspects was useful to find a variety of bugs at the system level (e.g., wrongly configured access permissions), cryptographic engines (e.g., broken AES mode), and even vulnerabilities in the included firmware (e.g., privilege escalation via system calls). Given the structured approach of our methodology, we could quickly write test-vectors and create bug reports and fixes. Our approach tends to abstract the core (code execution) and focus on the peripherals (accesses to memory mapped registers), therefore we missed most bugs in the core.

3.4 Research directions

Our approach to SoC security analysis centers around formulating hypotheses of potential weaknesses and verifying them with software. We believe that the use of software as a means of high-level expression facilitates cross-component and cross-layer evaluation, allowing us to try out different attacks on the system in short iterations. Generating hardware stimulus from the software abstraction levels is efficient to detect software-exploitable hardware vulnerabilities, at the price of loosing granularity and level of control on specific hardware blocks. Indeed, from software we do not have the freedom to stimulate all the inputs of a block, but only those that are exposed. On the other hand, software can easily drive multiple interconnected blocks at the same time. Visibility is instead preserved if using cycle-accurate simulation of the RTL code. While this approach does not cover all types of hardware bugs, it can be used by security experts without deep knowledge of the design, to overcome the limitations of conventional techniques [41]. However, the process still requires significant amounts of manual work and a high degree of security expertise. Moreover, the cycle-accurate simulation of the full system is very slow and it limits the potential application of more complex software-based approaches. In this section, we discuss how future research could alleviate some of the burden on the analyst and facilitate the execution of the tests, in particular by combining established approaches from the hardware-design community with ideas from software security testing.

3.4.1 On abstraction

During a security analysis, especially when searching for cross-component bugs, analysts attempt to regard the system as a whole where components can be verified together. We believe that flaws that affect the security of an SoC via cross-component interaction can best be discovered when the system is viewed from a perspective as abstract as possible, even if the implementation error that introduced the vulnerability is restricted to a single component: a block may function correctly when tested individually while still compromising the entire system's security in the interplay with other components.

We therefore think that software is the right layer for checking the security of the system as a whole. Software-based tests should enhance, not replace, the tests at the hardware layer that are already customarily performed during SoC development. In the following, we illustrate approaches that can potentially simplify software-based testing.

3.4.2 Generating tests

One task that currently requires manual effort is the creation of test software. Analysts need to read the security specification (if available), then often interpret it to obtain a more precise formulation of the desired security properties. Only then can they formulate hypotheses where the system may fail to meet the requirements, and finally devise corresponding test software. Note that, in the context of cross-component vulnerabilities, the security specification should remain at the high level of abstraction that includes all components of the system. Refining it to the level of individual components and their implementation is useful for component testing but undesirable for the purpose of assessing the security of the system as a whole.

In general, deriving meaningful tests from a specification becomes easier the more precisely the specification describes the system's expected properties. At the extreme, a machine-readable specification could automatically be translated to test cases [79]. Less precise descriptions leave room for interpretation and thus require expert knowledge to be used in security testing.

Once an actionable formulation of the security properties is available, it can be used to assess whether the implementation meets the requirements. Manually designed test programs, as used in our methodology, are only one option. In this context, it is worth mentioning symbolic execution and fuzz testing [19], both of which are popular approaches in software testing: they

check software by automatically exploring many possible paths through a program. However, while it is relatively easy to express security requirements in a software-only scenario, the same is not true when possibly faulty hardware components enter the picture [71]. We believe that a good specification could be used to drive the analysis carried out by such tools, helping with the difficulty of defining expected behavior. For example, symbolic execution could explore various interactions with the hardware, all the while checking that the security assertions put forth by the specification hold true in each tested case ([30, 62]).

3.4.3 Executing tests efficiently and effectively

Software-based tests have to be executed on some representation of the underlying hardware. Recent approaches for hardware simulation face the difficulty of scaling to increasingly complex designs [63]. In general, partitioning the system and describing blocks at different layers of abstraction helps to find good trade-offs among execution speed, the ability to catch low-level flaws, and simplicity of introspection and debugging. Partitioning and abstraction of hardware has been especially prominent in recent advances in dynamic binary firmware analysis. In the following, we show how these concepts map to the conventional hardware approach, and we discuss how SoC analysis can benefit from them.

To manage complexity, the traditional design and validation flow of electronic systems follows a top-down approach. Abstract specifications are iteratively refined, gradually introducing partitioning into separate components and implementation details [55]. At each iteration and layer, extensive validation is performed to ensure the correctness of the implementation. Partitioning allows testing a detailed component against blocks described at higher levels of abstraction. At the end, the components are integrated into a final product.

In contrast, many approaches developed for dynamic firmware analysis take the point of view of a security expert who analyses an already finalized commercial product, where part of the system may be unknown to the analyst. As a result, recent methodologies often use a bottom-up approach that reintroduces partitioning and abstraction. In this context, the CPU of an SoC is commonly emulated or completely replaced by a more abstract virtual machine, and the executed code is often translated into an intermediate representation at a desired level of abstraction [52]. However, other parts of the SoC under test (e.g., peripherals) are typically opaque to the analyst in this scenario and thus cannot be emulated. To overcome this issue, modern tooling allows either to specify models for the behavior of unknown parts

of the hardware [40, 69], or deploys near real-time forwarding mechanisms between the emulator and the real silicon for hardware accesses [58].

We believe that SoC testing and validation can greatly benefit from these two approaches—abstracting hardware components and selectively forwarding to real hardware while using an emulator—as shown in [62]. Additionally, the capabilities of existing tools (e.g., [69]) could be easily extended to cooperate with peripherals or other blocks at the RTL level. We present this approach in Chapter 6. Such extended tools would then serve as a natural platform for software-based security testing of SoCs. They were already designed with a focus on security and include a number of automated security checks (e.g., checks for memory corruptions). Moreover, they allow for more automated exploration techniques such as symbolic execution and fuzzing.

3.5 Conclusion

We have outlined how software-based tests are an effective approach for the security evaluation of an SoC. The software abstraction is very convenient, as it bridges the gap between the high-level security properties of the system and the low-level interactions across hardware components, as well as the gap between system and software security and hardware design and validation. Software-based approaches and tools are well known to software experts, and they are generally easy to set up starting from the final product, even without in-depth knowledge of the underlying hardware and without expensive simulation tools. Therefore, they could lower the entry barrier for analyzing the hardware components of an SoC, and facilitate the dissemination of knowledge across research communities.

Although this thesis mainly focuses on the challenges related to the dynamic analysis of firmware programs, in Chapter 6, we address the optimization of software-based methods to the hardware case. In particular, regarding test generation and automated design exploration, but also with the goal of efficient execution. Software security tools often take into account that analysts may not have full access to the design, as the hardware platforms running the software to be tested are often proprietary. However, these tools could greatly benefit from the availability of RTL code and models of the hardware components. The security and hardware communities could work together to create more and more accessible SoC platforms with representative vulnerabilities, to lower the barrier for developing new approaches and tools.

Chapter 4

Steroids: A fast, low-latency USB3 Debugger

4.1 Introduction

Embedded systems are becoming increasingly complex, connected and attached to online services. They are ever more present, especially with the rising of the so-called Internet-of-Things where today's products are attached to online services. However, this situation arises questions regarding their security? Recent literature repeatedly showed security issues with such systems. Nevertheless, there are a few tools available to analyze their security and existing solutions are generally limited by hindering factors. The specificities of embedded systems make techniques designed for traditionnal desktop applications difficult to apply.

Fuzzing is a common software testing method that demonstrates promising results for testing desktop applications. The core idea is to feed a program with malformed inputs and then monitor its execution in order to detect unexpected behavior (e.g., exceptions or crashes). This technique has proved its efficiency on desktop applications, however, results for embedded systems are hindered by limiting factors ([71]). One major challenge is the limited debug environment that often leads to silent vulnerabilities. In fact, vulnerabilities do not always lead to an external and observable events (e.g., exceptions, crash, unexpected outputs). To address this, fuzzing embedded systems requires instrumenting the firmware.

Firmware Instrumentation. Instrumentation techniques are common for advanced dynamic analysis techniques like fuzzing or sanitizers. These techniques use instrumentation to modify the code to retrieve coverage information or to add mechanisms to detect faults on time [69]. The instrumentation can be done using either dynamic instrumentation techniques, binary rewriting or at compile-time. Doppelgänger [38] uses binary rewriting to inject runtime integrity checks into an embbedded operating system. However, this method is difficult to apply. First, instrumenting binary code to add runtime memory protection mechanisms requires reconstructing the program semantic. In fact, types and data structures are lost during the compilation process. Without the source-code, this requires decompiling the binary code that is very difficult. Second, binary rewriting and compile-time instrumentation require inserting code precisely to avoid breaking time/space constraints that are relatively strong on some embedded systems. In addition, the heterogeneity of embedded systems makes these techniques difficult to adapt across architectures and requires a considerable amount of work to adapt the method for new devices.

Symbolic Execution consists of executing the firmware in a dynamic symbolic virtual machine while replacing concrete peripherals inputs by

symbolic inputs. However, this introduces other problems. First, the number of execution paths grows exponentially with the number of conditional branches depending on symbolic inputs. In practice, this technique is limited to the analysis of relatively small firmware. Second, context-insensitive analysis generally leads to large number of false alarms that are more difficult to filter for the analyst.

Partial-emulation. Another approach consists of re-hosting the firmware execution in a virtual machine [98], [93], [69], [90], [58]. This machine either simulates the Instruction Set Architecture or executes a semantic model while forwarding all the interactions with peripherals to the real device. This forwarding mechanism avoids arduous modeling of peripherals by using real peripherals. However, to interact with the embedded device, these techniques generally rely on remote debuggers. Moreover, debuggers are designed for interactive debugging by developers and are usually low performance.

In this chapter, we propose a novel debugger designed for automated embedded software testing. This debugger is designed to offer high performance and low latency to run existing testing tools on *Steroids*. Furthermore, it deals with embedded systems heterogeneity through flexible and re-programmable logic. We based our solution on off-the-shelf components to ease the reproducibility of our experiments. All the source-code (for hardware and software) is open-sourced to serve as a basis for future scientific contributions. We demonstrate the usefulness of our approach on AVATAR^2, a testing framework that orchestrates different security analysis tools. In addition, we compare *Steroids* performance with state-of-the-art approaches.

4.2 Previous Work

Debuggers play a major role in computer systems development by allowing developers to inspect their program as it is being executed. They are mechanisms to offer visibility and control over a program under test. Debuggers typically allow to to inspect any variables at any time during the tested program execution. Similarly, controllability refers to the ability to modify any variables at any time during execution. Debuggers can be divided into three categories.

Software-based debuggers are common on traditional desktop computers. Generally, they are specific software running on the same system than the software being debugged. Such debuggers are frequent for interpreted codes (e.g., Java Debugger), Instruction Set Simulator (e.g., QEmu

debugger), or OS-based applications (e.g., GDB). However, this approach is in practice difficult to apply for embedded systems. First, software-based debugger such as GDB requires OS-based capability that allows the debugger to access low-level information about another process. Without such an OS, the debugger has to deal with complex integration problems due to the co-existence of the debugger and debuggee in the same software context [99]. Furthermore, interactions with remote debugger require a communication channel that may be already busy.

In-Circuit Emulators (ICE). The ICE is a hardware tool that offers visibility into the internal operations of the device being tested. During development, the emulator replaces the target processor that often routes internal buses to output pins (i.e., bound-out chip). This emulator is attached to both the tested device and a host computer. Due to the tremendous cost of providing fast emulation memory, ICE are generally limited to relatively small microprocessors [46]. More complex micro-controllers generally integrate on-chip debugger.

An On-Chip Debuggers (OCD) is a debugging circuitry internal to the micro-controller. Generally, it offers generic debugging capability (e.g., breakpoints, watchpoints and memory inspection) and is accessible through a standard physical interface called JTAG that can be disabled on production chips , e.g., by blowing a fuse. Embedded systems generally adopt the standard IEEE 1149.1 that defines the Joint Test Action Group (JTAG) interface. The latter is an industry standard, widely used on embedded devices. It connects the internal debug logic of the chip to a JTAG probe. However, most existing JTAG debuggers [34] [81] [6] are based on USB 2.0 connection. USB 2.0 devices can transfer data on the bus only when an explicit request from the host controller has been sent (i.e. slave mode). For this reason, a USB 2.0 device has a relatively high latency.

Surrogates [58] introduces an efficient host-device debugger link. To achieve this, it uses an FPGA connected to the host through PCI Express and to the device through JTAG. Unfortunately, the hardware is not available anymore and the software has never been publicly released. Surrogates relies on DDC[1] a communication mechanism used with JTAG (using DCC registers), which is not available on modern ARM cores (i.e., Cortex cores). The DCC register was a feature of old ARM on-chip-debugger which has no equivalent on recent ARM CPUs. Finally, Surrogates requires a software stub to run on the device and does not offer basic debugging capability (i.e., breakpoints, watchpoints). Althought the Surrogates authors

[1]On ARM7TDMI, the DCC is a control register accessible to the processor and the debugger.

kindly shared their source code, extending it to support debugging capabilities would face to the same challenges than software-based debugger. This inflexibility, the lack of compatible hardware, and licensing problems, makes it difficult to adapt Surrogates to newer systems. Therefore, we were unable to replicate experiments with Surrogates and it wasn't possible to use it as a basis for research.

DStream [5] is a closed-source USB3-based debugger enabling remote debugging and instructions tracing of ARM-based chip. This system aims high performance. However, it only works with a proprietary and expensive Integrated Environment Development (IDE), making this solution unsuitable for open research. Furthermore, its integration with a third-party tool is difficult and not flexible as it requires communicating with the IDE's exposed interfaces (i.e., files). This layer of indirection increases the latency. Moreover, DStream has a closed hardware and software source-code making it very difficult to extend.

4.3 Design Objectives

After exploring the state-of-the-art, we seek to improve previous work in several ways. To this end, we select five desirable properties for the design of our solution.

1. Low-latency: minimize the duration time between the request emitted by a dynamic analysis tool (e.g., Avatar) and the response.

2. Open source: open as much as possible the design of our solution.

3. Off-the-shelf components to make our experiments easier to replicate.

4. Supporting fundamental features of debuggers (i.e., breakpoints, watchpoints, memory access).

5. Flexibility in the design to easily support more targets (On-chip debugger).

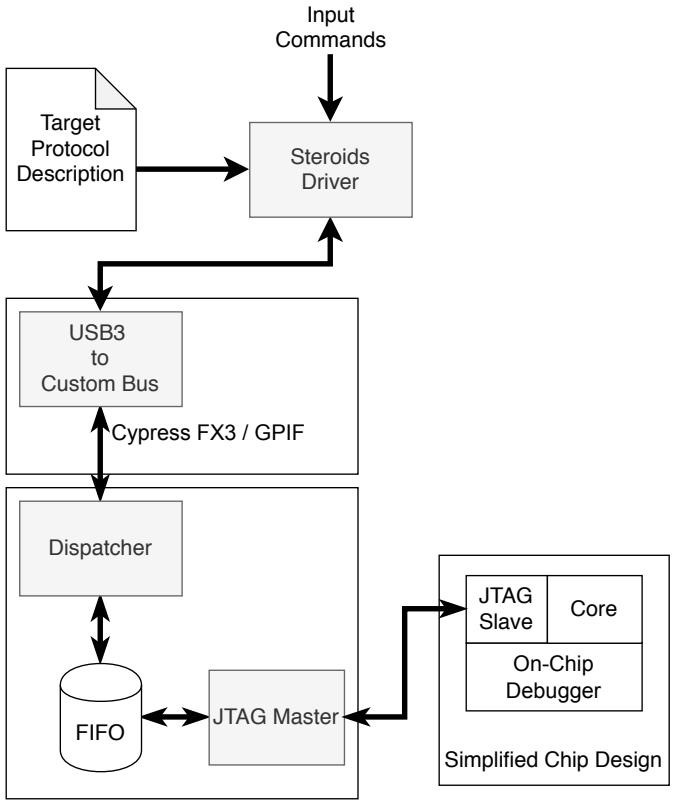


Figure 4.1: Overview of Steroids

4.4 Design Overview

In the following, we give an overview of the design of *Steroids* that seeks to respect our five desired properties. We provide a comparison with the previous work in Table 4.1. *Steroids* is a new, affordable, flexible, very high performance, and open-source debugger which is designed be used for security research on embedded devices. From a high-level perspective, we want to establish an end-to-end communication channel between a dynamic analysis tool on the host computer and the embedded system under test. This leads to the substantial question: which end-to-end protocol should we use?

End-to-End Protocol. General-purpose computer systems are often made of many independent peripherals that interact with each other using buses. Generally, this bus tends to be ever more generic with high throughput and low latency. Among them, PCI-Express and USB 3.0 are two good

candidates. PCI-Express offers lower latency than USB 3.0, however, it involves expensive and very specific hardware that does not fit our needs. Furthermore, even if the USB 3.0 latency is higher, our debugger is designed to interact with an on-chip debugger that generally admits a much lower bit rate. USB 3.0 interfaces are common nowadays, and there are available and affordable development kits.

	Surrogates [58]	DSTREAM [5]	Steroids
Host Interface	PCI-E	USB 3.0	USB 3.0
Target Interface	IEEE 1149.1(JTAG)	IEEE 1149.1 (JTAG)	IEEE 1149.1 (JTAG)
Low Latency	✓	✓	✓
Breakpoints,Watchpoints	✗	✓	✓
Access memory and peripherals on-the-fly	✗	✓	✓
Support ARMv7-M/ARMv8-M	✗	✓	✓
Do not require a Software Stub	✓	✓	✓
Open Source	✗	✗	✓
Open Hardware	✗	✗	✓

Table 4.1: Comparison of Steroids with the related work.

After having selected the underlying technology, we design our USB3-debugger. Figure 4.1 presents an overview of Steroids In particular, Steroids is composed of three main components.

1. A host driver. The driver exposes a high-level API to the dynamic analyzer. This API offers fundamental debugging features such as read/write memory, breakpoints, watchpoints. To deal with target device heterogeneity, a configuration file defines how the on-chip debugger works and how these high-level commands are translated to low-level pseudo-JTAG commands. This latter is a set of abstract commands that describe JTAG interactions with the device. These pseudo-JTAG commands do not reflect directly the JTAG protocol but instead express JTAG actions such as the values of the Instruction/Data Register and their size. Since they rely directly on the JTAG protocol, our solution supports any on-chip debugger that is accessible through a JTAG interface.

2. A USB 3.0 to FPGA bridge. This bridge receives pseudo-JTAG commands through a USB 3.0 socket. Then, it forwards these commands to an FPGA through a custom protocol. To fit the desired goal of having low latency, our bridge is implemented using programmable hardware, and therefore it does not involve any software interaction.

3. A Steroids commands dispatcher. The dispatcher is the master on the

bus with the USB 3.0 bridge. It gets notified when data is available by checking status signals. It balances the priority between receiving and sending data. For instance, it can decide to not receive data as its working queue is full or to prioritize the transmission of certain data to minimize latency. The latter are usually used to forward asynchronous interrupt signals as fast as possible. The bus between the dispatcher and the USB 3.0 bridge supports an addressing that reflect the USB 3.0 endpoints. This enables the dispatcher to forward the data to different internal logics. In our design, we expose two channels with two endpoints each (i.e., one for the host and one for the target). The first channel is for pseudo-JTAG commands while the second one is for interrupts.

4. A JTAG Master. The JTAG master receives pseudo-JTAG commands and operates like a processor. It decodes the commands and then executes them.

4.5 Implementation

In the following, we describe how we implemented *Steroids*. We now focus on the lower layers of the communication mechanism between the host and the real device.

In order to read and write the device memory, we directly connect to the system bus through the AHB-AP[2], which can be accessed with the JTAG protocol.[3] The AHB-AP port is available in ARM Cortex-based devices and allows a direct access to the peripherals. Inspired by Surrogates [58], we designed a custom device based on a Xilinx ZedBoard FPGA [42], to efficiently translate high-level read/write commands into low-level JTAG signals.[4] The FPGA is connected through a custom parallel port to a Cypress FX3 device [82] which provides an USB 3.0 interface. Unlike USB 2.0 where devices are slaves, USB 3.0 is a point-to-point protocol and, therefore, has a very low latency. With this setup we handle the burden of the low-level and inefficient JTAG protocol in hardware close to the device,

[2]The AHB-AP is a memory access port architecture that directly connects to an AHB based memory system

[3]An alternative would be a port using the faster SWD protocol, but this technology is less widespread than JTAG.

[4]Surrogates [58] was never open sourced but the authors shared their implementation. However, due to lack of hardware availability and other problems we eventually re-designed the debugger from scratch.

while we transmit high-level commands over a low-latency high-bandwidth bus to/from the host.

4.6 Evaluation

In the previous part, we have described how we implemented STEROIDS, a low-latency and high-performance USB 3.0 JTAG debugger. In this section, we evaluate the capability of our solution. In particular, we seek to answer two questions:

How long does it take to read/write memory using our debugger? In order to answer this question, we measured the duration time to read/write the memory device from the host computer. Then, we compared with the state-of-the-art (i.e., SURROGATES) and commercial debuggers.

How does it impact the performance of dynamic analysis tools? For this purpose, we added our driver on top of AVATAR^2 and evaluate its performance by comparing execution time on a sample from the official avatar repository.

4.6.1 Average I/O per Second

For our first experiment, we measure the duration time for reading/saving device memory through STEROIDS. We compare the result with the state-of-the-art: SURROGATES.

Protocol Details We proceed as follows. We measured the time to make 100,000 read/write requests to the device mapped registers. The device is an LPC1850 SoC. To compare with SURROGATES, we tune the JTAG frequency to 4 Mhz.

Observations and results *Steroids* has a read/write performance comparable to the fastest similar debugger (SURROGATES [58]). Using JTAG at 4 MHz, reads are 20% slower and writes are 37% faster (Figure 4.2). It seems that in our implementation the bottleneck comes from the USB software stack, rather than from JTAG, which can easily run faster, or from the USB protocol, which has itself a very low latency. Indeed, the GNU/Linux userspace library (`libusb-0.1-4`) performs system calls and DMA requests for each I/O operation, introducing a significant latency. Using bulk transfers of 340 reads is five times faster, since the latency for a USB operation appears only once. Unfortunately, code execution requires single memory accesses, but bulk tranfers could be used in many cases (e.g., state transfer

or DMA). SURROGATES uses a custom driver that exposes FPGA registers through MMIO over PCI-Express. Though the exact same approach is not possible, using a custom driver may improve STEROIDS performance.

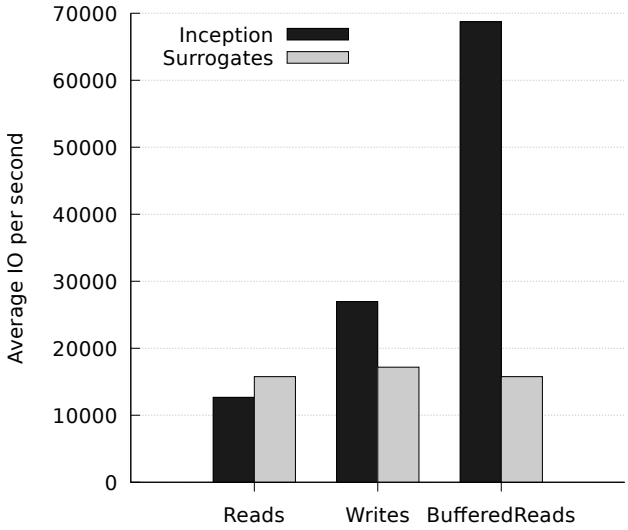


Figure 4.2: Average time to complete 1×10^6 read or write requests for SURROGATES and Inception (4 MHz JTAG). (libusb-0.1-4, Ubuntu16.04 LTS, Intel Corporation 8 Series/C220 USB Controller)

4.6.2 Improvment in Avatar

In this experiment (Table 4.2), we demonstrate the usefulness of *Steroids*. In particular, we verify our initial assumption where we identify the debug link as a bottleneck for existing partial-emulation solutions. Therefore, in this experiment, we measure the gain over AVATAR a well-known and open-source binary-based firmware analysis framework.

Protocol Details For the purpose of our experiment, we use a public example of AVATAR^2 called *nucleo_l152re*. This example runs a firmware binary on a Nucleo STM32L152RE device, then after reaching a certain point AVATAR^2 forwards the software context from the real device to QEMU. The execution continues in QEMU that forwards I/O to memory-mapped registers to the real devices. We measure the duration time to run 1000 times the state forwarding mechanism that copies 81920 bytes from the device memory to the QEMU memory. In addition, we measure the duration time to read/write 81920 bytes of the device memory. We report results in 4.2.

Observations and results The results of this experiment show an important performance improvement when using *Steroids*. In particular, the state transfer process is 16 times faster than the original approach (i.e., AVATAR^2 + STLink). This operation was introduced by AVATAR to reduce the slowdown of its full memory forwarding mechanism by executing code natively and then transferring the software state to another target where the code of interest is analyzed. However, without *Steroids* this mechanism is extremely slow, and it increases with the software context size. In addition, *Steroids* improves the read/write process performance by 16. These operations are frequently used during the firmware analysis and drastically limit the performance of the analysis on the original example. *Steroids* offers significant performance improvement that enables us to speed-up the analysis of complex firmware programs.

Operation	Data Rate [KB/s]	
	Avatar + STLink	Avatar + Steroids
Read	2.5	40.21
Write	2.49	72.02
Transfer State	2.37	40.06
Operation	Duration Time [s]	
	Avatar + STLink	Avatar + Steroids
Read	32.06	1.99
Write	32.07	1.11
Transfer State	33.77	2

Table 4.2: Average duration time in second to perform read/write/transfer-state operations on 81920 bytes repeated 1000 times for *Steroids* and AVATAR^2 (STLink debugger). Ubuntu18.04 LTS. 16GB RAM, I54500U 2.3GHz.

4.7 Conclusion

In this chapter, we introduced a high-performance and low-latency JTAG debugger. We demonstrate performance improvement over an existing hardware-in-the-loop/partial-emulation method called AVATAR^2. Furthermore, we demonstrate relatively similar performance than the state-of-the-art solution SURROGATES while basing our approach on off-the-shelf and affordable components. Contrary to SURROGATES that is not public and based on unavailable hardware, our experiments are fully reproducible. We believe, our solution is suitable for other research areas such as side-channel analysis (e.g., power analysis) or data-flow tracing.

Chapter 5

Inception: System-Wide Security Testing of Real-World Embedded Systems Software

In this chapter we introduce Inception, a framework to perform security testing of complete real-world embedded firmware. Inception introduces novel techniques for symbolic execution in embedded systems. In particular, *Inception Translator* generates and merges LLVM bitcode from high-level source code, hand-written assembly, binary libraries, and part of the processor hardware behavior. This design reduces differences with real execution as well as the manual effort. The source code semantics are preserved, improving the effectiveness of security checks. *Inception Symbolic Virtual Machine*, based on KLEE, performs symbolic execution, using several strategies to handle different levels of memory abstractions, interaction with peripherals, and interrupts. Finally, we integrate *Steroids* a high-performance JTAG debugger which performs redirection of memory accesses to the real hardware.
We first validate our implementation using 53000 tests comparing Inception's execution to concrete execution on an Arm Cortex-M3 chip. We then show Inception's advantages on a benchmark made of 1624 synthetic vulnerable programs, four real-world open source and industrial applications, and 19 demos. We discovered eight crashes and two previously unknown vulnerabilities, demonstrating the effectiveness of Inception as a tool to assist embedded device firmware testing.

5.1 Introduction

Embedded systems combine software and hardware and are dedicated to a particular purpose. They generally do not have the traditional user interfaces of desktop computers. Instead, they interact with the environment through several peripherals, which are hardware components that handle sensors, actuators, and communication protocols. The constant decrease in the cost of microcontrollers, combined with the pervasiveness of network connectivity, has led to a rapid deployment of networked embedded systems being used in many aspects of modern life and industry. These trends have greatly increased embedded systems' exposure to attacks. The consequences of a vulnerability in embedded software can be devastating. For example, the boot Read Only Memory (ROM) vulnerability used to jailbreak some iPhones cannot be patched in software, because the bootloader is hard-coded in the mask ROM without any patch mechanism [44]. Therefore, it is very important to thoroughly test such low-level embedded software. Unfortunately, the lack of tools, the intricacy of the interactions between embedded software and hardware, and short deadlines make this difficult.

Binary or source-based testing. The conditions under which testing

	FIE [39]	SURROGATES [58]	Avatar [98]	Inception
Using source code	✓	✗	✗	✓
Inline assembly	✗	✓	✓	✓
Binary code	✗	✓	✓	Some
Symbolic execution	✓	✗	✓	✓
Can use real peripherals	✗	✓	✓	✓
Early bug detection	✓	*n/a*	✗	✓
Fast forwarding	*n/a*	✓	✗	✓
Fast concrete execution	✓	*n/a*	✗	✓
Testing unmodified code	✗	✓	✓	✓
Low false positives	✗	*n/a*	✓	✓
Highly automated	✗	*n/a*	✗	✓
Open-source	✓	✗	✓	✓

Table 5.1: Comparison of Inception with the related work.

is performed can vary a lot depending on the context. The tester may have access to the source code, or just the binary code, and may use the device during testing or rely on simulators. Binary-only testing is frequently performed by third parties (pen-testing, vulnerability discovery, audit), whereas source code-based testing is more commonly done by the software developers or when the project is open-source. Access to source code provides many advantages; such as knowing the high-level semantics (e.g., the type of variables) of the program. This simplifies testing significantly.

An advantage of binary-only testing is that it can be performed independently of source code availability, and is, therefore, more generic. Indeed, even when source code is available, it can be compiled and the analysis can be performed on binary software. Unfortunately, this is inefficient, because during compilation, most code semantics are lost and this renders identification of memory safety violations and corruptions difficult. In fact, it has been shown that this effect is more severe with embedded software than with regular desktop software, due to the frequent lack of hardening of embedded software and hardware support for memory access controls such as memory management units [70]. Also program hardening (e.g., with Sanitizers [83]) is often impossible due to code space constraints and the lack of support for embedded targets.

Hand-written assembly. Unfortunately, the presence of hand-written assembly and third-party binary libraries is widespread in embedded applications. This severely limits the applicability of traditional source-based testing frameworks. There are two main reasons for the use of assembly language in embedded software development. First, although memory becomes cheaper and compiler efficiency improves, it is still often necessary to manually optimize the code (e.g., to fit in the cache, to avoid timing side-channels) and microcontrollers' memory size is still very constrained. Assembly is also necessary to directly interact with some low-level processor features (e.g., system-control or co-processor registers, supervisor calls).

Figure 5.1 highlights this problem on a set of sample programs from our test-suite (described in Section 5.4). Every sample contains at least one function with inline assembly. We further distinguish four categories of instructions, based on how they affect the system. From left to right: logical (e.g., arithmetic, logic), memory (load, store, barrier), hardware (supervisor call, co-processor registers access), control-flow (branch and conditional).

Logical and memory instructions are easy to translate to higher-level code. However, hardware *impacting* instructions strongly interact with the processor and affect the execution and the control flow. Common source-based frameworks cannot easily handle these low-level instructions. However, they are essential to handle tasks such as context-switching between threads. As a consequence, replacing those instructions with high-level code is difficult. We found that such instructions are present in all of the samples. Other places where assembler instructions or binary code is present is in Board Support Packages (BSP) provided by chip manufacturers or in library code directly present in ROM memory.[1]

Previous work. Table 5.1 summarizes the limitations of firmware security analysis tools. Avatar [98] and SURROGATES [58] focus on forwarding memory accesses to the real device, but only support binary code. Avatar relies on S2E [33] and, therefore, supports symbolic execution of binary code. On the other hand, FIE [39] tests embedded software using the source code, essentially adapting the KLEE virtual machine to support specific features of the MSP430 architecture. However, FIE does not try to simulate hardware interaction: writes to a peripheral are ignored and reads return unconstrained symbolic values. Moreover, FIE does not support assembly code which is very often present in such software and is, therefore, either entirely skipped or manually replaced by equivalent C code, if possible. This requires additional manual work, makes the state explosion worse, and leads

[1] For example, the NXP MC1322x contains drivers and a Zigbee software stack in a mask ROM [73].

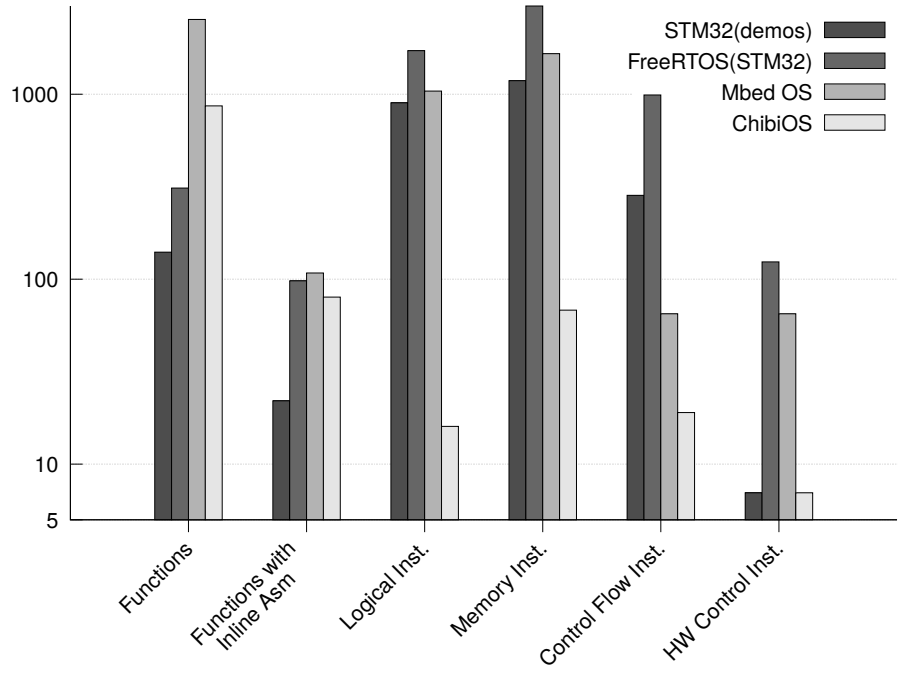


Figure 5.1: Presence of assembly instructions in real-world embedded software.

to a less accurate emulation.

Inception's approach. Inception's goal is to improve testing embedded software when source code is available, e.g., during development phases. We focus on the ability to perform security testing on complete systems made of real-world embedded software that contain a mix of high-level source code, hand-written assembly code, and, possibly, binary code (e.g., libraries). Unlike previous work, in Inception we preserve most of the high-level semantics from source code. We, therefore, can test software against real hardware peripherals with high performance and correct synchronization. Finally, to be broadly used, such integration tests need to be performed with a limited amount of manual work.

Contributions. In summary, in this chapter we present the following contributions:

- A new methodology to automatically merge low-level LLVM bitcode,

poor in semantic information and relying on the features of a target architecture, with high-level LLVM bitcode, rich in semantic information useful to detect vulnerabilities during symbolic execution

- A modified symbolic virtual machine, able to run the resulting bitcode code and to handle peripherals' memory and interrupts using different analysis strategies
- A thorough validation of the system to guarantee meaningful and reproducible results, and an evaluation of the approach on both synthetic and real-world cases
- A tool based on affordable off-the-shelf hardware components and source code that will be fully published as open-source

Chapter organization. The remainder of the chapter is organized as follows. Section 5.2 provides an overview of the approach and introduces the Inception tool. Section 5.3 presents the main implementation challenges and our validation methodology. Section 5.4 evaluates Inception on synthetic and real-world cases. Section 5.5 discusses limitations and future work. Finally, Section 5.6 concludes the chapter.

5.2 Overview of Inception

5.2.1 Approach and components

The main goal of Inception is to leverage the semantic information of high-level source code to detect vulnerabilities during symbolic execution, while also supporting low-level assembly code and frequent interactions with the hardware peripherals. Common symbolic execution environments usually run an architecture-independent representation of the code, which can be derived from the sources without losing semantic information. Alternatively, architecture-dependent binary code can be lifted to an intermediate representation that can be at least partially executed into a symbolic virtual machine, but that has lost the source code semantic information. These two cases differ greatly (e.g., in their memory model) and cannot easily coexist.

Inception solves the problem of coexistence by creating a consistent unified representation. In particular, Inception is composed of two parts. First, the **Inception Translator**, which generates unified LLVM-IR using a lift-and-merge process to integrate the assembly and binary parts of the program into the intermediate representation coming from the high-level

sources. This process also takes into account the low-level hardware mechanisms of the ARMv7-M architecture. Second, the **Inception Symbolic Virtual Machine**, which is able to execute this mixed-level LLVM-IR, and to handle interrupts and memory-mapped peripherals with different strategies, to adapt to different use cases. It can also generate interrupts on demand and model reads from peripherals' memory as unconstrained symbolic values. This VM is based on KLEE, a well-known open-source symbolic execution virtual machine which runs LLVM-IR bitcode. It provides high-speed access to the peripherals and could be easily extended for multiple targets.

In the following we give an overview of our lift-and-merge approach, of how KLEE performs security checks, and on how we extended it to support interrupts and peripheral devices.

5.2.2 Lift-and-merge process

Figure 5.2 shows the main stages of our bitcode merging approach and how source code with inline assembly (1) is transformed into a consistent bitcode (3) that can be executed by Inception VM. The example code contains the excerpt of a function written in assembly that requests a system call with `r0` holding a data byte.[2]

The rest of the code is composed of a main function, which calls the first assembly function, and the message to be sent. Using the appropriate LLVM front end (CLang for C/C++), source code (1) is translated into LLVM-IR bitcode. The resulting bitcode (2) shows that only C/C++ source code has been really translated into LLVM-IR. Indeed, the original purpose of LLVM-IR bitcode is to enable advanced optimizations before code lowering to the target architecture, whereas assembly is already at a low semantic level that cannot be represented or optimized by the LLVM compiler.

To solve this problem, we introduce a novel lift-and-merge approach, which we implement in *Inception-Translator*. This translator takes as input the ELF binary and the LLVM-IR bitcode generated by CLang. It generates a consistent LLVM-IR bitcode where assembly instructions have been abstracted to an LLVM-IR form. This step is done by a static lifter, which replaces each assembly instruction by a sequence of LLVM-IR instructions. We call the resulting bitcode a Mixed Semantic Level bitcode (mixed-IR), shown in (3), which contains:

[2]Figure 2 in the appendix shows the complete example, including the system call handler (in assembler) which sends the data byte over a UART by writing into the data register of the UART peripheral.

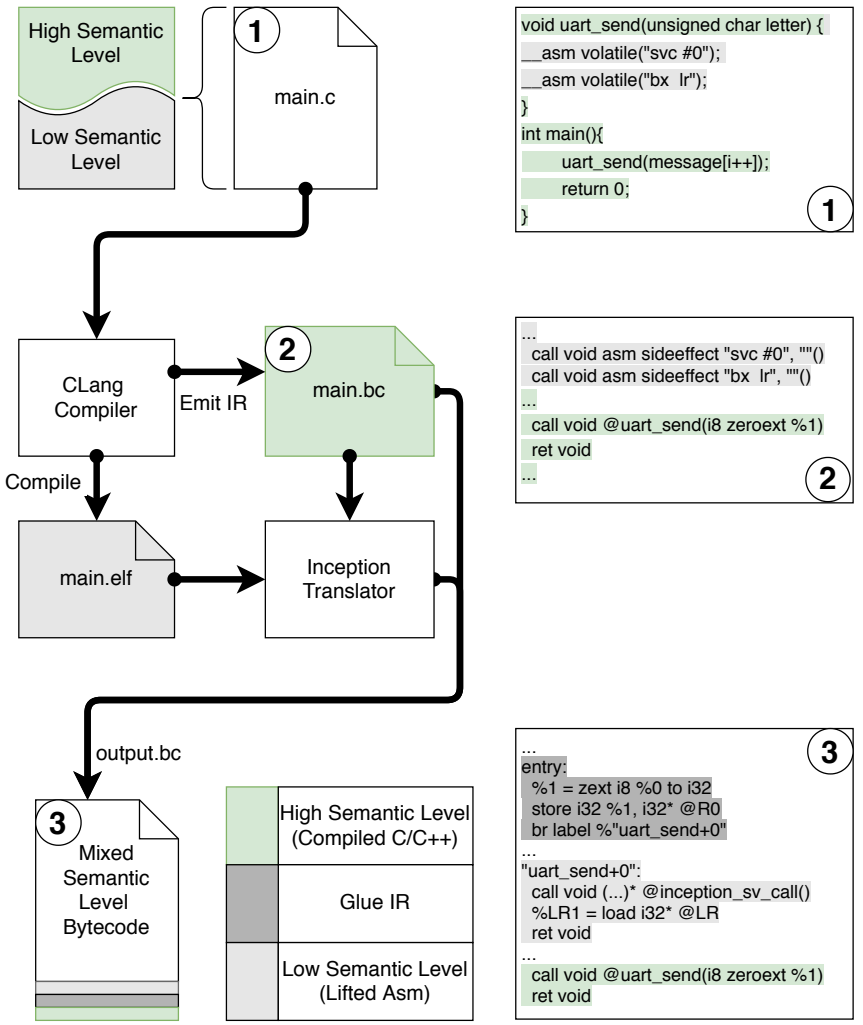


Figure 5.2: Overview of *Inception Translator*: merging high-level and low-level semantic code to produce mixed semantic bitcode. Excerpt of the translation of a program which includes mixed source and assembly.

High Semantic Level IR (high-IR) obtained from C/C++ source code. This is mainly the same code emitted by CLang, which has been augmented with external global variables that are defined in assembly source files. We reallocate these global variables in the IR.

Low Semantic Level IR (low-IR) deriving from assembly source code. This part is automatically generated by our static lifter. It contains

the translation of assembly instructions and some architecture-dependent elements that are necessary for execution. First, the CPU and co-processors' registers are modeled as global variables. Second, specific functions model the seamless hardware mechanisms that are normally handled by the CPU. For example, when entering into an Interrupt Service Routine (ISR), the processor transparently updates the Stack Pointer and it stacks a subset of CPU registers. When the ISR returns, the context is automatically restored, so that the code which was suspended by the interrupt can resume.

The Glue IR that acts as a glue to enable switching between the high-level semantics and the low-level semantics domains. This IR bitcode is generated by a specific Application Binary Interface (ABI) adapter, able to promote or demote the abstraction level. Indeed, communication and switching between layers mainly happens at the interface between functions, that is, when a high-level function calls a low-level one or the opposite.

5.2.3 Inception Symbolic Virtual Machine

The bitcode resulting from the lift-and-merge process is almost executable, but it still requires some extra support in the virtual machine. The main challenge is that high-IR accesses only typed variables and does not model memory addresses or pointers. On the other hand, the IR generated from assembly instructions has lost all information about types and variables, and only accesses pointers and non-typed data. Another challenge is handling memory-mapped memory, which is used but not allocated by the code, and interrupts and context switches, which are not modeled in KLEE.

To address these problems, we have extended KLEE with a *Memory Manager* and an *Interrupt Manager*. During (symbolic) execution the original *Memory Monitor* of KLEE performs advanced security checks on memory accesses. When a violation is detected, the constraint solver generates a test case that can be replayed.

The Memory Manager leverages the ELF binary and the mixed-IR to build a unified memory layout where both semantic domains can access memory. Specific data regions are allocated in order to run low-IR code, such as pointers contained in the code section, and some memory sections (stack, heap, BSS). Each memory address is configurable to mimic the normal firmware's environment. For example, a memory-mapped location could be redirected to the real peripheral, to prune the symbolic exploration and to use realistic values. Alternatively, it could be allocated on the virtual machine and marked as symbolic to model inputs from untrusted peripherals. Inception also supports Direct Memory Access (DMA) peripherals, provided that each DMA buffer is flagged as redirected to the real device

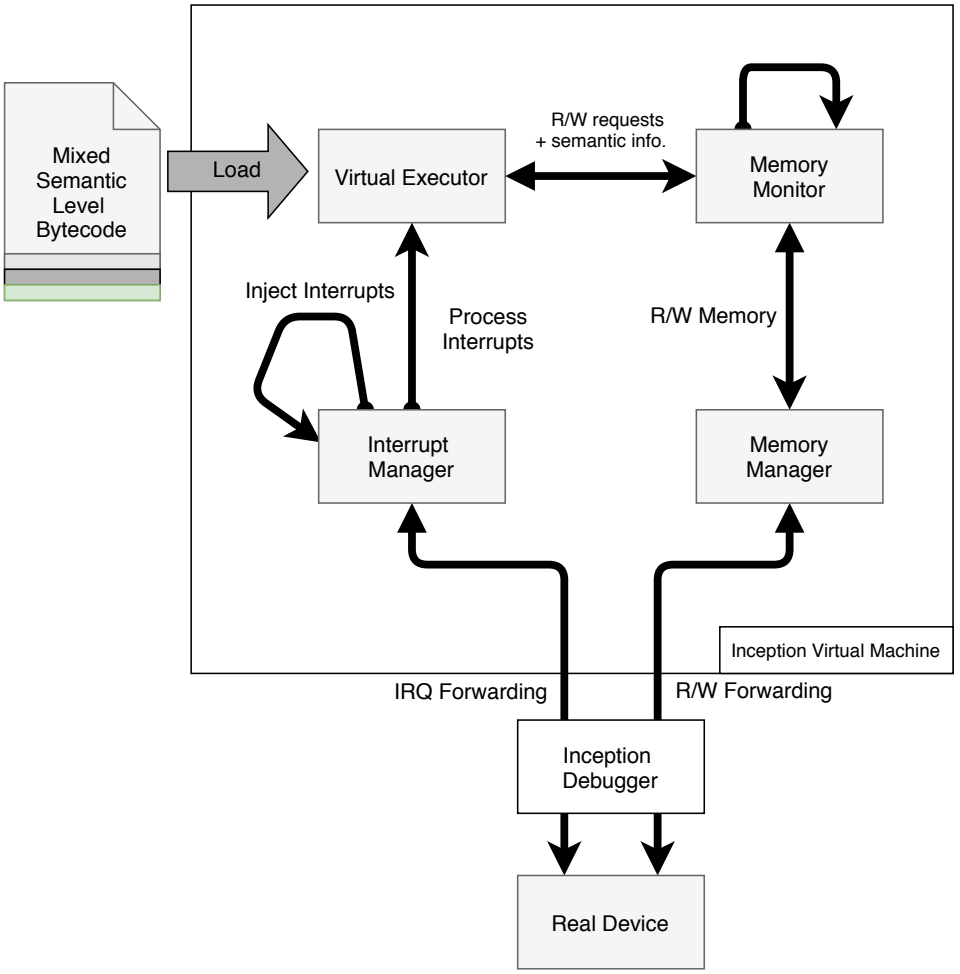


Figure 5.3: *Inception Symbolic Virtual Machine*, overview of the testing environment.

memory. Similarly to the other redirected locations, DMA buffers cannot hold symbolic values.

The Interrupt Manager gives KLEE the ability to handle interrupt events, by interrupting the execution and calling the corresponding interrupt handlers. Interrupt's addresses are resolved using the interrupt vector table. Interrupt events are either collected on the real hardware, or generated by the user when desired (by calling a special handler function). In the first case, the virtual machine and the real device are properly synchronized to avoid any inconsistency. We further extended KLEE to execute handlers that

switch the context between threads in multithreaded applications.

Memory Monitor and security checks. All security analyses mainly rely on the Memory Monitor of KLEE, which is able to perform security check for each access, based on the semantic information associated to it. The monitor observes the semantic information of the requests (requested type) and the semantic information of the accessed data (accessed type). When enough information is available, the monitor is able to detect memory access violations, e.g., out-of-bounds accesses, use-after-free, or use-after-return. Requests coming from high-IR, and accessing memory elements defined in high-IR, have enough information to detect most violations. On the contrary, requests that come from low-IR tend to have less information and a lower detection rate. However, thanks to the information coming from the high-IR, it is still possible to detect more problems than with binaries only.

5.3 Implementation and validation

5.3.1 Lift-and-merge process

In order to be able to glue assembly and binaries with source code into a unified LLVM-IR representation (mixed-IR), we apply two distinct processes.

The lifting process takes machine code (compiled assembly or binaries) and produces an equivalent intermediate representation (low-IR). This representation uses only low-level features of the LLVM-IR language and it mimics the original architecture (ARMv7-M), which contains some hardware semantics of the Cortex-M3 processor, such as the behavior of instructions with side effects. It is, therefore, (almost) self-contained, and a large part of it can be executed on any virtual machine able to interpret LLVM-IR. As explained in the following parts, we introduce some features to KLEE to make this code fully executable, in particular when dealing with context switches. Our lifter is based on three main components. First, a static recursive disassembler that finds all the instructions to translate and stores them into an internal graph representation. Second, a simple decompiler that reconstructs the control flow, including for indirect branches and complex hardware mechanisms (e.g., returns from interrupts and context switches). Finally, the lifter statically transforms a given machine instruction into a semantically equivalent sequence of LLVM-IR instructions. One important advantage of the static approach is that it enables further processing with the sources to produce mixed-IR. Moreover, it has a lower run-time over-

head compared to dynamic lifters that lift instructions during execution. Implementing all these components in a correct and reliable way requires significant engineering work[3], of which we omit most of the uninteresting details. In the next section we will describe some interesting aspects of the lifter.

The merging process takes the (almost) self-contained low-IR and the high-IR compiled from C/C++, to glue them together (with some glue-IR). This is the most challenging part, as they have different levels of semantic information and different views of memory. The first step is, therefore, to create a unified memory layout between the two IR-levels in the KLEE virtual machine. In addition to this, peripheral device addresses are made accessible in KLEE. The second step consists of identifying the best interface between the two representations and the mechanisms to exchange data at this boundary. We chose to use the Application Binary Interface (ABI) that regulates the communication between functions in a uniform way.[4] Our merger is able to generate glue-IR code that lets high-IR functions communicate with low-IR functions and vice-versa.

5.3.2 Unified Memory Layout

We now explain how we leverage both the lift-and-merge process and KLEE to create a unified memory layout. This memory layout is central for the low-IR and high-IR to coexist and communicate.

Processor registers are represented by global variables instead of LLVM-IR registers. In fact, the LLVM-IR is a Single Static Assignment (SSA) language, in which each instruction stores its result in a uniquely assigned register. These registers are not globally acessible. Therefore, LLVM registers cannot be used to represent CPU registers, which are assigned many times, and globally accessible by instructions.

The heap. Inception supports two dynamic memory allocation mechanisms. The first one is the native allocation function from the application (which can be written in assembly or C language). In this case, allocated

[3]We first used Fracture [59], a framework for lifting binaries to LLVM-IR. However, we eventually only reused a minor part of Fracture code. Indeed, Fracture's approach does not scale to all instructions, especially those interacting with hardware, and does not address the merging problem. Fracture was also designed for static analysis which did not need complete translation and is currently not maintained.

[4]Another option would be to set the interface at the native instruction level. An advantage would be to preserve most of the code translated from the high-IR in a function that includes only one inline assembler directive. However, the interfacing would depend on the compiler version and would be less robust.

variables lose semantic information and are encased in the heap memory region. This method is interesting for testing native allocation systems. However, it decreases the precision of corruption detection, because the heap memory is a container for indistinguishable contiguous variables, making it difficult to detect even simple out-of-bounds accesses. The second approach consists of replacing the native allocation functions by KLEE's own allocator. KLEE allocator was specifically designed to detect memory safety violations. In particular, KLEE isolates each allocated variable with a fixed-memory region (the *red zone*). Even though this mechanism does not detect all violations, any access to this zone will be detected as a memory corruption. Another advantage of KLEE allocation is that it can detect memory management errors such as invalid free of local or global variables.

The normal klee stack is used when high-IR code is running. Each function has its own function frame object, which contains metadata about the execution. This includes information about the caller, the SSA registers values (which hold temporary local variables), and the local variables (which are allocated using the normal KLEE mechanism). A separate **stack** is used by the low-IR code. This stack is modeled as a global array of integers, allocated by the memory manager at the same address and size than the .stack section of the symbol table. Variables in this stack are not typed. However, the ABI adapter mechanism presented in the next section allows different IR levels to access variables on both stacks.

The Data region contains mixed semantic-level variables. Indeed, when the high-IR allocates data, the resulting memory object is typed and allocated at the same address as indicated by the symbol table, to keep the compatibility with assembly code. On the other hand, data can be defined by the assembly code and accessed by high-IR. In this case, we use the semantic information present in the external declaration of the high-IR to allocate a typed object. The third possible case is data allocated by assembly code, but never accessed by high-level code. In this case no semantic information is present, and allocation depends on the information from the ELF symbol table.

5.3.3 Application Binary Interface adapter

Low-IR functions follow the standard Arm Application Binary Interface (ABI) [16], whereas high-IR functions follow the LLVM convention. Therefore, whenever the *Static Binary Translator* finds a call or return that crosses the IR levels, it invokes the *ABI adapter* to generate some glue-IR that adapts parameters and return values.

When a high-IR function calls a low-IR function, the high-IR arguments

(typed objects) must be lowered to the architecture-dependent memory (stack/CPU registers). In the opposite case, stack and CPU registers must be promoted to high-IR arguments. Similar considerations apply to return values. This process is similar to serializing and deserializing the LLVM typed objects, to store them as words in the LLVM variables that represent the CPU registers and the stack, where they are used by low-IR. Note that during serialization the types are lost, but deserialization is still possible thanks to the high-level information present in the source code. For example, consider an assembly function that passes a `struct` by value to a C function. Knowing the size and address of the destination, the adapter generates the glue-IR that copies CPU registers and stack words from the low-IR to the high-IR destination. Another example is an assembly function that returns a pointer. In low-IR, the pointer is stored as a simple integer word in the `ro` register. Since the adapter knows that the expected return type is a pointer, it can write the glue-IR that performs the cast to it. All main C types are supported. There are four possible connections between low-IR and high-IR (code examples available in the appendix in section .1):

1. **High-IR to low-IR parameters passing**. A glue-IR prologue takes the input arguments from the KLEE stack (where the high-IR caller stored them) and brings them to the CPU registers and/or low-IR stack (where the low-IR callee expects them).

2. **Low-IR to high-IR return value**. A glue-IR epilogue takes the return value (stored in `ro` by the low-IR callee) and promotes it to a typed object in KLEE stack (used by the high-IR caller).

3. **Low-IR to high-IR parameter passing**. Before calling the high-IR function, some glue-IR takes the input arguments from the CPU registers or the low-IR stack (where the low-IR caller stored them) and promotes them to typed objects on the KLEE stack (used by the high-IR callee).

4. **High-IR to low-IR return value**. Just after the high-IR callee returns, some glue-IR moves its return value from the KLEE stack to `ro`.

5.3.4 Noteworthy control-flow cases

We focus on the explanation of noteworthy control-flow instructions and hardware mechanisms to show their impact for the security checks. We

omit the details for the other instructions.[5]

Control-flow instructions. The main challenge when dealing with control flow consists in finding a good mapping between high-level control flow operators present in LLVM-IR (e.g., `call`, `if/else`) and low-level ARMv7-M instructions, which are at a lower abstraction layer (they directly modify the program counter, and sometimes rely on implicit hardware features).

We translate to an LLVM call instruction any Arm instruction that saves the program counter before changing its value (i.e., direct and indirect branch-and-link instructions) to an LLVM call instruction. In order to support indirect calls, we leverage an optimization technique called *indirect call promotion* [15, 61, 26, 86]. This technique consists in transforming each indirect call into direct conditional branches and direct calls. Indirect call promotion has been introduced to improve the performance of branch prediction [15]. Conditional branches compare the target address of the indirect call with the entry point of each possible function in the program. If the condition is true, this function is called directly. This is equivalent to enforcing a weak control flow integrity policy, and akin to what KLEE already does for C/C++ function pointers. It would be possible to enforce stricter control flow integrity checks by retrieving the control flow graph with a static analysis or a compiler pass.

We translate all instructions that restore the previous program counter, for example `bx lr` and `pop pc`, to return instructions. These returns still work as intended even if the return address is corrupted. However, we do not rely on side effects (return to a corrupted address) to detect corruption. We rather detect the corruptions by relying on the memory checks, e.g., to detect buffer overflows.

We implement all other direct (conditional) branches and `it-blocks`[6] with simple direct branches available in LLVM-IR.

Interrupts and multithreading. The control flow of the program is also modified by interrupts, which asynchronously block the normal execution and call-defined handler functions. Interrupts are used very frequently in embedded programs to synchronize the peripherals with the embedded software in an event-driven fashion, or to implement multithreading.

Inception VM can receive interrupts from the real device (when real peripherals are used and generate interrupts) or generated by the user using helper functions (e.g., to stress specific functions in a deterministic way).

[5] The lifting of these instructions is similar to re-implementing a Cortex-M3 in LLVM-IR based on the ARMv7-M reference manual.

[6] In ARMv7-M an "`it-block`" is a group of up to four instructions executed only if the condition of a preceding `it` instruction is true.

We extended KLEE so that the main execution loop checks for the presence of interrupts to serve. In this case, KLEE executes an LLVM-IR helper function that accesses the interrupt vector table in the firmware memory to resolve the address of the interrupt handler to call, based on its identifier (ID). This dynamic resolution is necessary only if the firmware overwrites the vector table. If the vector is fixed, a slight speedup in execution can be obtained by storing the vector in a configuration file, loaded by KLEE at startup.

Before giving control to an interrupt handler, and when returning from it, a Cortex-M3 processor performs several seamless operations (e.g., stacking and unstacking the context, managing two stack modes). In Inception, a special glue-IR helper function generated by our lift-and-merge process performs these steps.

To implement multithreading, operating systems such as FreeRTOS heavily rely on the interrupt and stack management features. In summary, the operating system, which has its own stack, manages a separate stack for each thread. Context switching is possible because when a thread is interrupted, its context is saved to its stack, and the context of the resuming thread, including the program counter, is pulled from another stack. The switch is done in part by the processor and in part by the operating system. Inception fully supports this process, since all the required features are self-contained in the mixed-IR. *Inception VM* extends KLEE's call stack management, to be able to handle one call stack for each thread. Briefly, whenever a new thread is spawned, a new call stack structure is generated and assigned to it.

Synchronization with the real device. To collect interrupts on the real device, we insert a stub on the device that registers one handler for each possible interrupt. When an interrupt is fired, the handler is called and notifies KLEE thanks to the forwarding system. The main challenge of this architecture is to keep the virtual machine and the device synchronized, without inconsistencies and race conditions, even in presence of multiple priorities. This needs to be done carefully and uses several mechanisms. In particular, the interrupt handler on the device should not return until the corresponding KLEE handler terminates. This is necessary, for example, to mask interrupts with the same or lower priority until the handler ends, as it happens in the real device, and to avoid the flooding of new interrupts.

A complete example. Figure 5.4 shows an example of context switch triggered by an interrupt generated on the device. On the right we see how the identifier of the interrupt is used both to notify KLEE at the beginning and to acknowledge the stub at the end. The acknowledgement is per-identifier, so that the stub can be interrupted by higher priority interrupts.

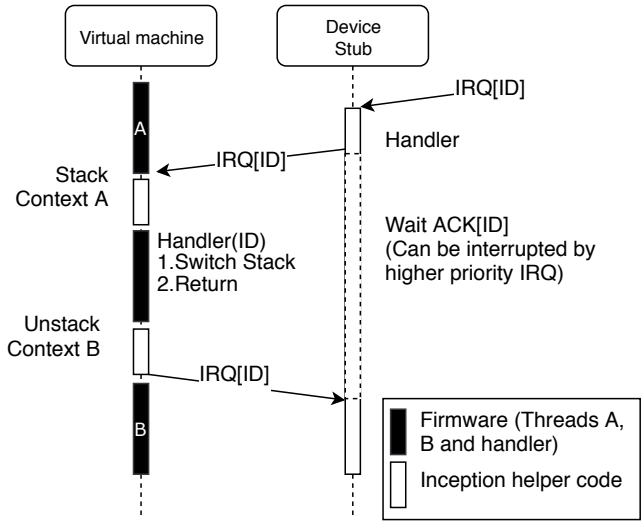


Figure 5.4: Context switch due to an IRQ.

On the left, we can observe the switch between threads enabled by the seamless context stacking and unstacking.

In summary, *Inception* fully handles interrupt synchronization with the host virtual machine thanks to *Steroids*, while previous work had only limited interrupt support [98].

5.3.5 Forwarding mechanism with Steroids

In the previous parts we described how we integrated peripheral devices and interrupts in the virtual machine. We now focus on the lower layers of the communication mechanism between the host and the real device.

In order to read and write the device memory, we directly connect to the system bus through the AHB-AP, which can be accessed with the JTAG protocol.[7] The AHB-AP port is available in Arm Cortex-based devices and allows a direct access to the peripherals. Inspired by SURROGATES [58], we designed a custom device based on a Xilinx ZedBoard FPGA [42], to efficiently translate high-level read/write commands into low-level JTAG signals.[8] The FPGA is connected through a custom parallel port to a

[7] An alternative would be a port using the faster SWD protocol, but this technology is less widespread than JTAG.

[8] SURROGATES [58] was never open sourced but the authors shared their implementation. However, due to lack of hardware availability and other problems we eventually re-designed the debugger from scratch.

Cypress FX3 device [82] which provides an USB 3.0 interface. Unlike USB 2.0 where devices are slaves, USB 3.0 is a point-to-point protocol and, therefore, has a very low latency. With this setup we handle the burden of the low-level and inefficient JTAG protocol in hardware close to the device, while we transmit high-level commands over a low-latency high-bandwidth bus to/from the host. Our debugger (i.e., *Steroids*) is able to communicate with the stub running on the device and handle interrupts using a dedicated asynchronous line and shared memory locations.

Handling Interrupts. Forwarding interrupts is more complex because they are asynchronous events and close to the hardware. Indeed, interrupts are generated on the device and partially processed by the interrupt controller (NVIC) and the core before being serviced by a user-defined interrupt handler. For example, the NVIC orders the interrupts in terms of priority and the core can mask them. In our system, we use the real NVIC on the device, configured by the software as a normal peripheral. We also add a special forwarding command to write the control register of the core responsible for masking, which is accessible via JTAG. Finally, we use dummy interrupt handlers that are executed on the device. These handlers have a dual function. First, they catch the interrupt events which they send to the host via the forwarding mechanism. Second, they execute at the same time as the handlers in the host virtual machine, keeping host and device synchronized in the same state. As long as the handler is executing on the host, a corresponding dummy handler is executed on the device. This is fundamental, as the way NVIC and core handle new incoming interrupts also depends on the execution of the handler. For example no other interrupt of the same or lower priority can be served while the handler is still executing, whereas an interrupt with higher priority will interrupt the current one. To be precise, there is a small time window during which the device has already an active interrupt and the virtual machine not, due to the transmission latency. This does not introduce inconsistency, providing that any read of the active interrupt register by the software returns the state of the virtual machine and not the one of the device. While Avatar introduces the idea of an interrupt stub to collect interrupts, it does not solve these synchronization problems in detail. While prior work had basic interrupt support [98], this mechanism allows us to support high speed interrupts and complex priorities, which was not possible before. To sum up, we can handle high-speed interrupts with complex priorities, and we correctly synchronize the real device with the host virtual machine, whereas previous work had only limited interrupt support [98]. Once the dummy handler is called, it notifies the FPGA using a handshake protocol based on two general purpose

input/output signals (GPIOs). The main command execution loop on the FPGA is interrupted, the FPGA reads the interrupt number in memory location written by the handler and forwards it to the host through the normal system. During the communication with the FPGA interrupts are masked to guarantee atomicity, but in any other moment the dummy handler can be interrupted by higher priority interrupts. Once it has sent the event to the FPGA, the dummy handler enters a loop, waiting for the handler on the host to terminate. When the host handler returns, the system notifies the dummy handler by writing a flag in memory. There is one location for each interrupt number, to support nesting of higher priority interrupts. We plan to explore the use of a trace port instead of GPIOs to send interrupt events, however, the dummy handlers would still be necessary for synchronization.

In summary, we provide a clean design for an efficient, cheap , and open-source solution, which can be used to experiment and replicate research that requires customizable debuggers (e.g., [74]).

5.3.6 Validation

We carefully validated Inception to obtain a reliable tool.

Regression Tests. We created a framework for automated regression testing of the code. Around 53200 tests are performed at several levels of abstraction, from unit tests up to tests involving all components. Results are compared to a *Golden model* (i.e., a known and trusted reference). For example, we compared single instructions against the real Cortex-M3 processor, assembly functions against the C code from which they originate or alternative implementations, and complete applications against their behavior on the native hardware. We stress symbolic execution on known control flow cases, and bug detection on known vulnerabilities.

Arm Cortex-M3 lifter. The correctness of the lifter is particularly important to obtain correct execution. Our framework generates all possible supported instructions, starting from a description of the instruction set. Then, for each type, it creates several tests with random initialization of registers and stack. Finally, in executes them both on the device and in Inception, and it compares the final state of registers and stack. Table 1 in the appendix summarizes all the tests we performed.

5.4 Evaluation and comparison

After validation, we evaluated Inception over a set of interesting samples, which we explain in this section. We first focus on the effects of seman-

tic information on vulnerability detection and on the speed performance of the tool. Then, we show analyses on more complex examples including, for example, assembly code for multithreading and statically linked libraries. Finally, we explain how Inception found corruptions in three industrial applications under development, including a boot loader. Evaluating and comparing tools for embedded software analysis is hard because of the lack of an established benchmark suite. This is rendered harder due to the large number of different hardware platforms. While some of the examples we use below are proprietary, we also built a large set of validation and evaluation examples, sometimes based on existing open-source code. Those examples will be made available together with Inception and may provide a basis for such a benchmark.

5.4.1 Vulnerability detection

Detection rate at different semantic levels. We evaluate how vulnerability detection is affected by the semantic level of high-IR and low-IR and their interaction. In particular, we explore if KLEE can detect memory corruptions on a vulnerable path, depending on how variables are allocated and accessed by different types of IR. Our analysis samples are based on the *Klocwork Test Suite for C/C++*[9], which includes out-of-bound, overflow, and wrong dynamic memory management errors. We initially compile them to high-IR (and binary). We then selectively force the decompilation from binary to low-IR of some functions, obtaining 40 different interaction cases. Table 5.2 summarizes the different combinations of allocation and access of memory objects at different semantic levels, and the corresponding detection results, which we comment in the following.

[9] https://samate.nist.gov/SRD/view.php?tsID=106

Table 5.2: Overview of memory checks between LLVM code at different IR semantic level.

Allocation		C with KLEE Allocator		C Native Allocator		ASM or Binary	
		Accessed from					
		C	ASM	C	ASM	C	ASM
Dynamic Allocation	Check Types	✓	✓	✗	✗	✗	✗
	Red Zone	✓	✓	✗	✗	✗	✗
	Heap Consistency Checks	✓	✓	✗	✗	✗	✗
Stack Allocation	Check Types	-	-	✓	✓	✗	✗
	Red Zone	-	-	✓	✓	✗	✗
.Data or .BSS Allocation	Check Types	-	-	✓	✓	✗	✗
	Red Zone	-	-	✓	✓	✗	✗
Not Allocated Memory	KLEE Detection	-	-	✓	✓	✓	✓

First, detection works only for those memory objects allocated in high-IR for which we have semantic information. However, the memory accesses can come from both high-IR and low-IR or be related to the return value of low-IR functions. For example, a C function allocates a buffer that is then

improperly used by an assembly function. If the called function overflows the buffer, it will access an unallocated memory space of the high-IR domain where memory objects have a defined size, type and which are separated from each other by a red zone. The semantic information of high-IR memory objects greatly improves the detection of vulnerabilities even if it occurs in low-IR code. However, if the buffer is allocated by a low-IR code (assembly or binary code), the lack of semantic information about the variable prevents the detection of the overflow. The same mechanism is applied to local (static) allocation and global allocation.

Second, when using KLEE *dynamic allocation functions, all vulnerabilities can be detected in both high-IR and low-IR,* whereas if we use some implementation in the code of the application, the detection rate drops to almost zero for both high-IR and low-IR. However, in this case we can test the code itself of the allocation functions, either in high-IR or low-IR depending on the case.

In summary, in 40 synthetic tests, 70% of the inserted vulnerabilities were found and no false vulnerability was reported.

Comparison with binary-only approaches. When testing embedded binary code, it is hard to catch memory corruptions because of the lack of semantic information, code hardening, and operating system protections. For example, [70] highlights the problem when fuzzing a STM32 board, and it uses several heuristics to catch corruptions. To compare this approach with Inception, we analyze the same firmware (EXPAT XML parser with artificial vulnerabilities). Each vulnerabilty (stack/heap-based buffer overflow, null pointer dereference, and double free) has its own independent trigger condition. We start with the source code compiled to high-IR, but we also generate cases with low-IR by forcing the decompilation of vulnerable functions. To use Inception, we mark the input as symbolic and run the samples with a timeout of 9.0×10^1 s. Results are visible in Figure 5.5. Our approach successfully uses all the semantic information available, keeping a good detection rate even in presence of some low-IR code. We could integrate the heuristics from [70] to improve results even further. One of the vulnerabilities could be detected, but it is not triggered because of state explosion (47k states) and the constraint solver (using 67.5% of the time), which are problems inherent to symbolic execution and common to KLEE.

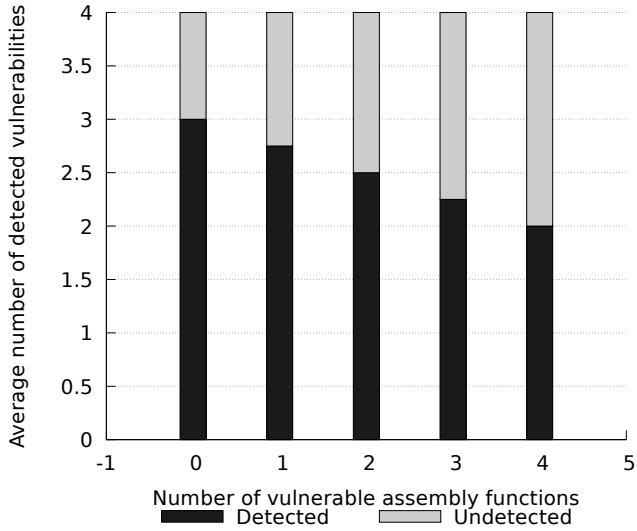


Figure 5.5: Evolution of corruption detection vs. number of assembly functions in the EXPAT XML parser (4 vulnerabilities [70], symbolic inputs, and a timeout of 9.0×10^1 s).

5.4.2 Timing overhead

Overhead of the executor. We evaluate the execution speed of the virtual machine using the DHRYSTONE[10] v2.1 benchmark, compiled without any optimization in LLVM-IR. Inception has 38% of slowdown overhead compared to KLEE, but if we disable the multithreading support the overhead becomes insignificant. Inception is 17 times slower than the real hardware[11]. This is mostly due to execution in the KLEE virtual machine.

Overhead of low-IR (advantage of high-IR). One of the advantages of our source-based approach is that we maximize the use of high-IR, which is more compact and faster than low-IR. To provide a rough example, we force 3 functions out of 12 in DHRYSTONE V2.1 to be translated from binary, which is a realistic proportion. This adds 343 more IR lines to the initial 1636, reducing the speed by around 43%. Low-IR does not seem to affect the time spent in the constraint solver. For example, we run `bubble sort and insertion sort`, with a symbolic array of 10 integers and a timeout of 9.0×10^1 s. Both the high-IR and the low-IR versions spend about 90% of

[10] DHRYSTONE is a synthetic computing benchmark program, available at http://www.netlib.org/benchmark/dhry-c.

[11] Value reported by the manufacturer for a STM32 with Cortex-M3.

the time in the constraint solver.

Benchmark of some real applications. We evaluate the overall performance (software stack and forwarding) of three popular protocols: ICMP, HTTP, and UART. For the first two, we use the `Web`[12] example for the `LPC1850` board. We use the Ethernet interface of the real device, forwarding memory accesses and interrupts. In particular, we identify the DMA buffers and configure Inception to keep them on the memory of the real device. For the UART, we use the driver of the `STM32` board, again using the real peripheral. For all protocols we use simple clients (`ping`, `wget`, and `minicom`) on a laptop, and we repeat measurements for 100 runs. Results are shown in Figure 5.6. There are two reasons why ICMP and HTTP are slower than UART. First, they have a more complex software stack. Second, they require forwarding of many interrupts and of large DMA buffers.

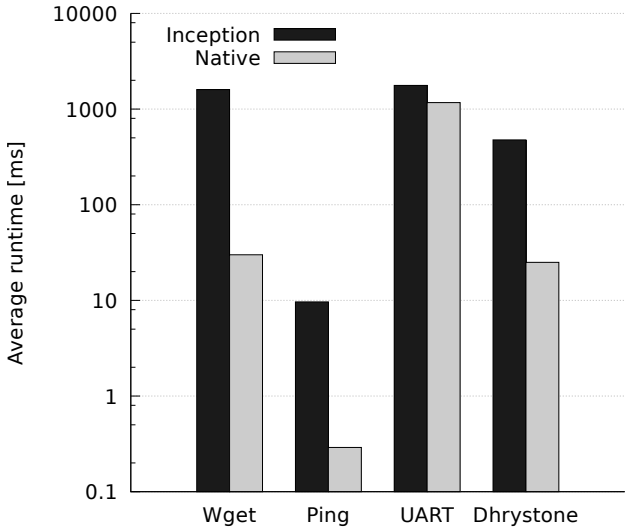


Figure 5.6: Performance comparison between native execution and Inception. (libusb-0.1-4, Ubuntu16.04 LTS, Intel Corporation 8 Series/C220 USB Controller)

5.4.3 Analysis on real-world code

We evaluate the capabilities of the Inception system on two publicly available real-world programs. These two samples cover the different scenarios

[12]It is part of the `lpc1800-demos` pack available at https://diolan.com/media/wysiwyg/downloads/lpc1800-demos.zip

Type	Total	Detected	Rate
Division by Zero	88	88	100%
Null Pointer Dereference	131	131	100%
Use After Free	62	62	100%
Free Memory Not on Heap	1.131	1.131	100%
Heap-Based Buffer Overflow	38	38	100%
Integer Overflow	112	0	0%
Total	1.562	1.450	92%

Table 5.3: Corruption detection of real-world security flaws based on FreeRTOS and the Juliet 1.3 test suites.

in which Inception can be applied.

FreeRTOS is a market-leading real-time operating system supporting 33 different architectures.[13] It provides a microkernel with a small memory footprint and thread support. For this, it uses small assembly routines that strongly interact with the features of the target processor and it is, therefore, a good test case for Inception. We show that Inception can execute low-level functions that deal with multithreading before reaching vulnerable areas. We experiment with the injection of vulnerabilities in one thread, symbolic execution with producers and consumers, and corruption of the context of a thread.

We take the injected vulnerabilities from the NSA Juliet Test Suite 1.3 for C/C++, which collects known security flaws for Windows/Linux programs.[14] We selected tests related to divide by zero, null pointer dereference, free memory not on heap, use after free, integer overflow, heap-based buffer overflow. We skip tests that cannot run on our target STM32L152RE (e.g., those that require a file system or a network interface) and those that the LLVM 3.6 bitcode linker cannot handle (poor support of the C++ name mangling feature) for a total of 10384 and 1214 deletions, respectively. Furthermore, we update namespace names to comply with CLang 3.6. We obtain 1562 tests which we embed in FreeRTOS threads.

To trigger the vulnerabilities, Inception has to first execute low-level code containing assembly, and in some cases also to flag as symbolic the output of a software or hardware random generator. The interrupts required for context switches and timers can be either collected on the real device or simulated (with the appropriate generation functions). We chose the

[13] https://www.freertos.org/

[14] https://samate.nist.gov/SRD/around.php#juliet_documents

second option to be able to run many tests quickly. We set a timeout of 3.00×10^2 s and we observed that we can reach these regions without manual effort or modification to the multithreaded code (Table 5.3). The detection rate is 100% for divisions by zero, null pointer dereference, use after free, free of non-heap allocated memory, and heap buffer overflow vulnerabilities. Integer overflows are not detected at all in KLEE (version 1.3). However, we note that in general it may be possible to detect a consequence of the overflow later.

We also wrote a simple multithreading library that uses the same hardware features as FreeRTOS. On top of it, we created a simple example with three threads, where two consumers use the data put in a circular buffer by a producer. This simulates, for example, an application that processes sensor data. Depending on a symbolic value, threads execute in different order with different data. Inception can easily find a condition that triggers an overflow in the circular buffer. We also simulate the presence of a vulnerable code that corrupts the context of a thread, in particular its program counter on the stack. In this case, when the corrupted thread resumes, Inception detects that the program counter is invalid (not part of a thread that was correctly started before). Note that they may be false positives (if such behavior was intentional) or negatives (if the corrupted address is still valid).

libopencm3 is an open-source library that provides drivers for many Cortex-M devices.[15] We test some examples in which the library is a statically linked binary. It is very similar for *Inception Translator* to lift and merge a function in a statically linked library or from a function that contains inline assembly. For example, we write a sample that uses the CRC peripheral to compute the Code Redundancy Check (CRC) on a buffer. The CRC peripheral computes one word at a time, so the driver iterates over the buffer locations. Besides this, the application calls other `libopencm3` functions to initialize the STM32 device and to configure and blink LEDs. Though the driver and the other functions are translated from the binary, the buffer is part of the application code written in C; therefore, we have semantic information on its type and size. Similarly, Inception knows the memory layout and the location of the other variables. If the low-IR driver is called with an incorrect length parameter, this leads to an out-of-bound access which is detected by Inception. Similarly, if the buffer is dynamically allocated and erroneously freed, Inception detects a use after free. The semantic information used for detection would not have been exploited by a binary-only tool.

[15] https://github.com/libopencm3/libopencm3

5.4.4 Usage during product development

Commercial bootloader. Bootloaders are good targets for Inception, since they contain low-level code and they often parse untrusted inputs. Moreover, they are hard to test when the real hardware is not available yet and tests on prototypes may be not accurate. To show the potential of Inception in these conditions, we analyzed a bootloader under development, and we found a problem that would have been difficult to detect on FPGA-based prototypes.

Our target is a secure bootloader with several options, stored in a One Time Programmable (OTP) memory. When it executes, the bootloader holds in SRAM a structure containing some information about the application (e.g., start address, stack address). This structure is pointed by `p_header` in the pseudo-code that follows:

```
1  void start(){
2    switch(boot_modes) {
3      case NO_SECURE_BOOT:
4        context.p_header->start_addr = FLASH_MEM_BASE;
5        context.stack = SRAM_STACK;
6        jump_to_application();
7      break;
8      case SECURE_BOOT:
9        do_secure_boot();
10     break;
11     default:
12       error();
13   }
14 }
```

To prepare the analysis, we configured Inception with the memory layout of peripherals. We also flagged the OTP memory as symbolic, to explore all possible paths deriving from different boot options. Despite the lack of hardware, Inception did not require any change to the source code. During symbolic execution, Inception detected a corruption (write to an invalid address) at line 4, and the solver gave us a test case to reach this condition. We manually inspected the code and confirmed that the `p_header` pointer is not initialized.

In summary, the bootloader writes a value to an address held in a non-initialized SRAM location. If the invalid write does not trigger other errors, the bootloader can still execute and successfully load the application at `start_address`, making this problem hard to detect. In particular, it does not crash on the FPGA prototype, because `p_header` is null (SRAM zeroed at reset), which is mapped to writable memory. A write to 0 would instead produce a memfault on the real device, as 0 would be mapped to a read-only

memory. Bug had only been detected later in the development process, like on silicon, it would have been very expensive to fix it. Indeed, it would have required costly re-design and re-fabrication.

From a security perspective, an attacker may at least partially control the value of `p_header`. For example, we could imagine a scenario in which certain options lead to writing to this location, and a fast reboot preserves it (SRAM is not initialized). Besides changing the destination before the write, an attacker could change it after, so that the bootloader would dereference a wrong `start_address` at which to load the application.

Chip SDK. We tested a Software Development Kit (SDK) for a commercial chip, at a stage when a prototype of the hardware was not even available yet. Therefore, we configured reads to peripherals to return unconstrained symbolic values. Inception found a test case in which a bit-wise shift depended on an untrusted value (overshift), which we confirmed by manual inspection. In this case, the error leads to the wrong configuration of a peripheral and unexpected behavior. More generally, overshifts could lead to overflows or out-of-bound accesses. Early detection is useful to avoid expensive fixes later.

Commercial payment terminal To show the potential of Inception when hardware is available, we tested a payment terminal under development, using the FPGA prototype to redirect most peripherals and their interrupts. The application communicates with an external smart card through a card reader, which we mark symbolic since it is not trusted. This mix of concrete and symbolic peripherals effectively explores the code, avoiding state explosion. Inception found eight potential vulnerabilities (out-of-bound accesses), that have been reported to developers and still have to be confirmed.

5.5 Discussion

In the following we discuss the advantages and limitations of Inception.

Application vs. (software/hardware) environment. The key to using symbolic execution in realistic settings is to limit the expensive symbolic exploration to a small critical code region, treating the (software/hardware) environment separately. S2E [33] investigates how different strategies to cross this partition affect the analysis. Inception offers several options. Dynamic allocation can be either part of the environment (host functions with concrete or concretized inputs), or part of the code under test (where symbolic values can propagate). The former reduces the symbolic space at the price of completeness, whereas the second one preserves complete-

ness at the price of higher complexity. A peripheral can be treated as a stateless untrusted function that ignores inputs and returns unconstrained symbolic values. This leads to the exploration of all possible paths, also those that would not be globally feasible with the real peripherals (making false positives possible). Though useful for drivers when the hardware is not yet available, this option does not scale because of state explosion. Alternatively, Inception can use the real peripherals with concrete values, reducing the problem. Globally unfeasible paths are reduced too, but they could still appear if the states of peripheral and code become inconsistent (e.g., if symbolic execution switches state during the access pattern to a stateful peripheral). However, symbolic exploration visits the higher-level logic of the application rather than the drivers, making the problem less common. A more thorough study is left as future work. A complete testing of a firmware program would require considering interrupts at any single instruction, which in practice is not feasible. Previous work [76] reduces the frequency of timer-based interrupts by executing them only when the firmware goes in low-power interrupt-enabled mode. However, this solution can miss issues that may occur when interrupts are processed during the firmware execution. Inception enables users to generate interrupts on demand that are useful to obtain deterministic sequences or to stress the code, but it is neither complete nor guaranteed to try cases that are actually possible. Collecting the interrupts from the real hardware covers realistic cases without additional complexity, but suffers from possible inconsistencies as explained for peripherals. We plan to analyze enable/trigger patterns to detect which symbolic states must serve an interrupt when it arrives.

Semantic gap. Inception increases the overall vulnerability detection rate for applications containing assembly parts because it is able to preserve as much as possible of the semantic information. However, the detection level for the bitcode generated from low-IR could be improved, for example, reconstructing typed objects from assembly, using DWARF debug information, and adding extra detection heuristics (e.g., from [70]).

Support for binaries. Even though Inception targets the analysis of source code during development, binary code may appear as a precompiled library (e.g., we have encountered this case with `libopencm3`). Since the binary is statically linked with the application, Inception can collect enough information about function prototypes, symbols, and their addresses to successfully decompile and merge the library functions used by the application. This case is handled not much differently from that of functions containing inline assembly.

Support for C/C++. Inception supports all main C types but inherits

from KLEE the support for symbolic floating-point values. Regarding C++, we support the C subset. Name mangling is poorly supported by the LLVM 3.6 linker, and the syntax of some namespaces is not accepted by the Clang 3.6 front end, which is more strict than GCC 4.8. The subset that works in Inception is generally enough for embedded software and for our samples.

Manual effort. Inception reduces the manual effort required for analyzing embedded software, since it does not require any change to the original code to support assembly and peripherals. The main challenge for a user is the general problem of tuning symbolic execution. On a more practical side, Inception requires extending compilation to CLang (e.g., in presence of GCC-specific features) and to extract the memory layout of mapped memory from the datasheet. This can be at least partially automated with custom or existing tools. Moreover, compiling with CLang is worthwhile to profit from its advanced static checks.

Lifter and its validation. The way we validated Inception's lifter is similar to the validation of the ARMv7-M formal instruction set [45] or to the testing of CPU emulators [65]. Using a machine-readable architecture specification to generate the lifter [78], or to generate test cases, would provide a higher level of assurance. However, none of the current formal descriptions for Arm processors [77, 45] support the ARMv7-M architecture. Lifters are often used for particular applications. For example, PIE [36] relies on S2E to perform static analysis, whereas FirmUSB [53] lifts binary code to perform symbolic execution. Research in lifter design is quite active. Fracture [59] tries to leverage the semantic information already present in compilers in the other direction. This approach is successful for generating bitcode for static analysis, but we found it unsuitable for generating executable LLVM bitcode and for integration with our merging step. Other approaches [86, 53, 17, 25, 50] are based on static translation, while tools such as QEMU [22] use dynamic translation, which we avoid, since integrating them with our merging approach would be complex.

5.6 Conclusions

In this chapter we highlighted the need for handling programs as a whole in embedded systems development and testing. Like prior work, our experiments show that testing based on the source code leads to a much better bug-detection level than when working only on the binary code. These two constraints together imply that embedded programs need to be considered with both their high-level source code and their hand-written assembler code. For this purpose, we compile plain C functions with LLVM

toolchain into LLVM-IR and functions which include assembler into native code, which we then directly lift to LLVM-IR. Finally, we merge this code and execute it in Inception VM (a modified KLEE), which handles both abstraction levels and is able to interact with the hardware using a fast debugger (STEROIDS). We performed extensive tests and found two new vulnerabilities and eight crashes in embedded programs, including bootloaders which were written to be included on a Mask ROM. The entire project is open-sourced to make our results easily reproducible and available at https://github.com/Inception-framework/.

Chapter 6

HardSnap: Leveraging Hardware Snapshotting for Embedded Systems Security Testing

Advanced dynamic analysis techniques such as fuzzing and Dynamic Symbolic Execution (DSE) are a cornerstone of software security testing and are becoming popular with embedded systems testing. Testing software in a virtual machine provides more visibility and control. VM snapshots also save testing time by facilitating crash reproduction, performing root cause analysis and avoiding re-executing programs from the start.

However, because embedded systems are very diverse, virtual machines that perfectly emulate them are often unavailable. Previous work therefore either attempt to model hardware or perform partial emulation (see Chapter 5), which leads to inaccurate or slow emulation. However, such limitations are unnecessary when the whole design is available, e.g., to the device manufacturer or on open hardware.

In this chapter, we therefore propose a novel approach, called HardSnap, for co-testing hardware and software with a high level of introspection. HardSnap aims at improving security testing of hardware/software co-designed systems, where embedded systems designers have access to the whole HW/SW stack. HardSnap is a virtual-machine-based solution that extends visibility and controllability to the hardware peripherals with a negligible overhead. HardSnap introduces the concept of a hardware snapshot that collects the hardware state (together with software state). In our prototype, Verilog hardware blocks are either simulated in software or synthesized to an FPGA. In both cases, HardSnap is able to generate HW/SW snapshot on demand. HardSnap is designed to support new peripherals automatically, to have high performance, and full controllability and visibility on software and hardware. We evaluated HardSnap on open-source peripherals and synthetic firmware to demonstrate improved ability to find and diagnose security issues.

6.1 Introduction

From automotive to house appliances, embedded systems are becoming ever more present in our modern life. The semiconductor market is highly competitive with a very short time-to-market, in particular for micro-controllers addressing niche markets. Moreover, embedded systems complexity is growing, making them more difficult to verify. The security of embedded systems is a big concern. First, fixing security problems is often difficult. Silicon-based hardware cannot be patched. Some firmware programs are often also stored on read only memory (e.g., mask ROM) and is similarly impossible to modify. Fixing such problems generally requires expensive redesign and fabrication steps and therefore increases the time-to-market. Second, em-

bedded systems are hidden away in more complex systems such as phones, computers, payment terminals or system controllers, and therefore are subject to storing personal data, drive physical systems or part of complex industrial plants. Third, the growing connectivity and attachment to online services, make them more exposed to attacks. For all these reasons, there is an important need for security tool to test embedded systems before production.

Virtual machine introspection has formed the basis of many dynamic analysis methods such as coverage-guided fuzzing [64, 101, 100], symbolic execution [27, 18], forensics analysis [51, 85, 68], and malware analysis [95, 75]. This approach offers a full visibility and controllability over the system under test, thus enabling sanity checks, tracing, concurrent testing, and coverage measurement. Similarly, research on dynamic analysis of embedded systems tends to use emulators to gain in visibility and controllability of the execution. However, research in this field generally faces limited performance, difficulty to automate peripherals interactions, and limited introspection on hardware peripherals (i.e., visibility or controllability).

Hardware Peripherals Interactions. When re-hosting embedded systems in a virtual machine environment, one recurrent challenge is the difficulty to correctly handle hardware interactions. In fact, embedded systems are purpose-built computers, mixing specific hardware peripherals and firmware programs. There are typically many interactions between firmware and peripherals which occur frequently during firmware execution. As a consequence, the analysis of firmware without proper peripherals interaction is often impossible.

Modeling Hardware Peripherals. Depending on the context, hardware peripherals are modeled using different approaches. When hardware design source code such as peripherals' Hardware Description Language (HDL) is not available, the behavior of those peripherals needs to be replicated. Previous efforts replaced peripherals with hand-written [22] or automated [49, 14, 35] behavioral models. However, these methods are error-prone, time-consuming and difficult, especially for complex peripherals. To avoid peripherals modeling or partial emulation, hardware-in-the-loop schemes forward Input/Output to the real device [98], [58], [37], [93]. Despite the gain in performance and automation, this method significantly limits the visibility and controllability of peripherals that cross the boundaries of the virtual machine. In particular, this makes complete state snapshotting impossible: part of the state of the system is in the hardware. If the tester can have access to the peripherals source code, it provides him

many advantages. This enables peripherals to be either simulated or emulated on an FPGA. Simulation offers a high visibility and control over the overall simulated hardware blocks. However, HDL simulation suffers from a significant performance slowdown. Another solution consists in emulating the peripherals on an FPGA that may run at a speed similar to that of the silicon chip. Contrary to a simulator, FPGA does not offer a high visibility/controllability on the running design. All these solutions make system snapshotting challenging, and therefore limits the performance of existing firmware testing methods.

System Snapshotting. Snapshots are useful to replicate events (e.g., corruption) for more detailed analysis. They also improve analysis performance. This is typically interesting for symbolic execution and fuzzing engines that may use snapshotting techniques to reduce the overhead of re-executing the program from zero when concurrently testing multiple paths of a program. Tools combining partial-emulation and symbolic execution generally break the virtual machine boundaries, and therefore, introduce significant consistency problems mainly due to the difficulty to control peripheral state. One obvious solution to this problem would be a record-and-replay approach, however, it is extremely slow and error-prone as the number of interactions to replay may be considerable and time sensitive. Talebi et al. [93] report 8800 I/O operations just for the initialization of the camera driver in the Nexus 5X. Replaying all the interactions would consume a significant amount of time. Alternatively, when the HDL is available a logic model [88, 32, 47] can be automatically generated. The resulting model is accurate and offers a full-visibility and control over the simulated design. Unfortunately, simulators have an important performance penalty that slowdowns the dynamic analysis.

In this chapter, we introduce hardware state snapshotting, a mechanism to save and restore hardware state, which extend traditional software and VM snapshotting to hardware-in-the-loop snapshotting. We implement this technique in HardSnap, our framework based on a symbolic virtual machine, based on Inception (see Chapter 5), to co-test hardware and software. HardSnap was designed for performance, automation, full-visibility, and full-controllability over the whole design under test (firmware and hardware). In particular, we combine symbolic software execution and hardware emulation targets (i.e., FPGA and HDL simulator). HardSnap, further enables analysts to easily drive hardware components, express security properties using a high level of abstraction, or test firmware programs. Using its symbolic execution engine, HardSnap can be used to generate software test vectors to test hardware. HardSnap also makes possible to clone the hard-

ware state between different targets to get the best of each world (FPGA performance vs. full traces in a simulator). HardSnap can be either used for testing the whole design or only a sub-system. We believe this would facilitate its integration in a product development flow where components and firmware are build concurrently.

Since our methodology aims at assisting hardware/software designers, we evaluate it on a complete system. Unfortunately, despite the growing presence of open source hardware, there are no complete SoC and firmware which we could reuse for testing. We therefore demonstrate the capability of our tool on a synthetic design composed of open-source hardware peripherals and firmware. We argue that this is a realistic scenario, as such components are commonly used on commercial microcontrollers.

Contributions. In summary, in this chapter we present the following contributions:

1. A system-wide co-verification framework that supports hardware and firmware analysis. This framework generates new test cases thanks to a symbolic execution engine.

2. A novel methodology to save/restore embedded system state including hardware peripherals and firmware program. Our method automatically insert introspection mechanisms in hardware peripherals. This enables hardware state observation and control at any time. We propose two methods based on a simulator and an FPGA, to get the best of each world.

3. A novel multi-target support for hardware emulation enabling state transfer at any time during the analysis to get the best of each hardware targets.

6.1.1 Related Work

Research in dynamic analysis of embedded systems has been an active topic over the previous decades. This has led to different approaches that we can group in four main categories. They are presented in table 6.1.

Full Emulation. This approach relies on full-system emulation to mimic the behavior of the original machine. A compelling example of such approach is S2E [33] that is based on QEMU [22]. S2E enables symbolic execution while emulating peripherals through behavioral models written in

[10] No hardware interactions.

[11] Using either a sub-approximation or an over-approximation.

	Over-Approximation	Sub-Approximation			Full Emulation			Partial Emulation			Simulation	Hybrid	
	FIE [39]	P²IM [14]	HALucinator [35]	Pretender [49]	(FPGA) Analyzer Logic Integrated [97] [96]	FPGA	S2E [33]	Avatar1/2 [98, 69]	Inception (in Chapter 5)	Surrogates [58]	Verilator [88]	QEmu+System-C [32]	HardSnap
Abstraction Level	B[10]	B[10]	B[10]	B[10]	P	P	B	B/P	P	P	L	B/L	B/L/P
Symbolic Execution	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✓
Full Visibility	✗[11]	✗[11]	✗[11]	✗[11]	✗(Limited scope)	✗	✓	✗	✗	✗	✓	✓	✓
Full Controllability	✗[11]	✗[11]	✗[11]	✗[11]	✗(Limited scope)	✗	✓	✗	✗	✗	✓	✓	✓
Ensure HW/SW Consistency	n/a	n/a	n/a	n/a	n/a	n/a	✓	✗	✗	n/a	n/a	n/a	✓
Automated Peripheral Modeling	✓	✓	✓	✓	n/a	✓	✗	✗	n/a	n/a	✓	✗	✓
Fast Forwarding	n/a	n/a	n/a	n/a	n/a	n/a	✓	✗	✓	✓	n/a	✓	✓
Open-source	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓

Table 6.1: Comparison of HardSnap with the related work. L: Logical (RTL level) P: Physical B: Behavioral

C. It snapshots the entire emulator program to offer full visibility, control and ensure consistency during the symbolic execution that is able to concurrently and exhaustively explore multiple execution paths. S2E enables full-system analysis (i.e., peripherals and firmware), however, it requires hand-written behavioral model for peripherals that is not easy and error-prone.

Partial Emulation. To address the problem of supporting hardware peripherals automatically, AVATAR [98] redirects hardware interactions to the real device. This method has later been followed by SURROGATES [58] and INCEPTION (see Chapter 5). The former and the latter support advanced analysis of embedded systems thanks to a dynamic symbolic execution (DSE) engine. However, contrary to S2E they do not ensure hardware/software state consistency during the entire analysis because of the lack of control and visibility on the real device. In fact, the real hardware peripherals are accessed concurrently by many software states (one by execution path), changing the internal state of peripherals. The result may lead to inconsistent states (unrealistic values) affecting the dataflow and control flow, and therefore, leading to false positives or false negatives.

Automated Re-Hosting. Previous efforts replaced peripherals with automated models. Peripherals are replaced by either an over-approximation [39] or a sub-approximation [49, 14, 35]. These methods have limitations. First, the approximation of the interactions with the underlying hardware may lead to false positives (i.e., using not realistic values) or false negatives (i.e., all the realistic values are not considered). Second, they limit the visibility to the tested software only, and therefore make bug analysis challenging when they are related to hardware components. These methods address analysis of firmware programs when the peripherals source code is not available.

Simulation. Hardware simulators [88], [9] generally transform the Hardware Description Language (HDL) into a cycle-accurate behavioral model that is tested using RTL or software-driven testbench. Contrary to the silicon chip, cycle-accurate simulators offers full visibility and control over the hardware, enabling DSE to generate snapshots and ensure consistency during the analysis. However, hardware simulation is slow. Moreover, peripherals (accelerators) are often designed to accelerate complex and slow computations, simulation of such peripherals is very slow. To overcome this performance limitation, the HDL can be synthesized to run on an FPGA. Nonetheless, FPGAs offer a limited visibility over the design making snapshotting difficult.

Hybrid. To get the best of both worlds, researchers sought to mix

different approaches. A tool combining simulation and emulation has been developed by Chiang et al. [32] to enable cycle-accurate and full emulation. This method offers a full visibility and control over the hardware, however it does not perform advanced dynamic analysis.

6.2 Motivation

In the following, we give details about the motivation behind this work.

How do peripherals affect firmware execution? Embedded systems are purpose-built computers mixing hardware peripherals and firmware programs. There are different reasons for the presence of peripherals. They may offer an interface to the external world (e.g., actuators and sensors), a inter-device communication interface (e.g., UART and wireless communication), a hardware accelerator (e.g., cryptographic accelerators) or internal resources (e.g., interrupt controller, Direct Memory Access, Memory Protection Unit). Peripherals affect the firmware data-flow and control-flow in different ways. Generally, firmware programs read inputs from peripheral through a Memory-Mapped IO or a Port-Mapped IO. Peripherals can also modify the system memory (DMA). Moreover, the firmware execution can be interrupted by the peripherals when a task completes. Those important interactions between firmware and peripherals make firmware execution dependent on the hardware. Additionally, bugs may originate from these interactions. For all these reasons, co-testing hardware and firmware is important.

How does snapshotting reduce the overhead of re-execution? Snapshots enables a program under test to be revived at an earlier point. This is typically interesting for symbolic execution and fuzzing engines that may use snapshot techniques to reduce the overhead of re-executing the program from zero when concurrently testing multiple paths of a program. As observed by Muench et al. [71], fuzzing embedded systems requires to restart the target under test after each fuzzing input to reset a clean state for further test inputs. Without HardSnap, restarting the embedded systems requires a complete reboot of the device which is extremely slow. For symbolic execution, snapshots are heavily used. Each time the symbolic engine executes a branch where the condition is symbolic, it forks the entire program memory in two states (one snapshot for each part of the condition). Then, the analysis explores all paths concurrently according to the state exploration heuristics. This approach requires intensive snapshot reload. While traditional symbolic execution keeps all the tested system within the virtual machine boundaries, symbolic execution with hardware-in-the-loop

breaks this assumption. This may lead to inconsistency (e.g., peripherals and software state mismatch) or extremely high overhead due to the need to re-execute the program from zero.

Inconsistency due to incomplete snapshots. When peripherals offer limited controllability and visibility, it is almost impossible to generate a complete snapshot of the embedded system. This limitation leads to different scenarios for dynamic analysis of embedded systems. For the sake of clarity, we illustrate these scenarios with a simple use case that we present in Fig. 6.1. In this use case, a firmware program consists of two different execution paths that request a specific computation to a unique peripheral. In return, this peripheral emits an interrupt signal to notify that the computation is done. Then, the firmware executes the corresponding interrupt request (IRQ) that reads the result from the peripheral. We identify three different approaches for co-testing hardware and firmware programs. First, the naive-and-consistent approach tests firmware execution path one after the other, and it ensures a clean state by rebooting the entire system and restarting the execution from the program start. This approach is often adopted by fuzzer [71]. Unfortunately, it may involve a significant number of time consuming reboots. Furthermore, it re-executes code having the same effect for different execution paths (e.g., the INIT sequence), this is not efficient. Second, the naive-and-inconsistent approach tests different execution paths concurrently. This approach is the one adopted by hardware-in-the-loop-based DSE [98, 37]. These tools evaluate concurrently different execution paths of the firmware under test while forwarding I/O to the real device. The resulting analysis improves performance over the previous method, however it introduces inconsistencies. In fact, if the same hardware is driven in parallel by different software execution path, thus leading to erroneous output values and execution flow. In our example, the routine 'REQ A' and 'REQ B' are executed concurrently. In result, the peripheral receives data emitted by the routine 'REQ B', and it aborts the computation of 'Task A'. The control flow gets affected since only one of the two interrupts is emitted by the hardware. This is in fact a simple example, but in reality, the naive-and-inconsistent approach may lead to complicated inconsistencies affecting complex control flow and data flow of the embedded system. These inconsistencies drastically affect the analysis correctness by introducing false positives and false negatives. Finally, our approach, called HardSnap, enables hardware/software snapshotting. This snapshot avoids any time-consuming reboot, and it enables consistent concurrent analysis of firmware programs.

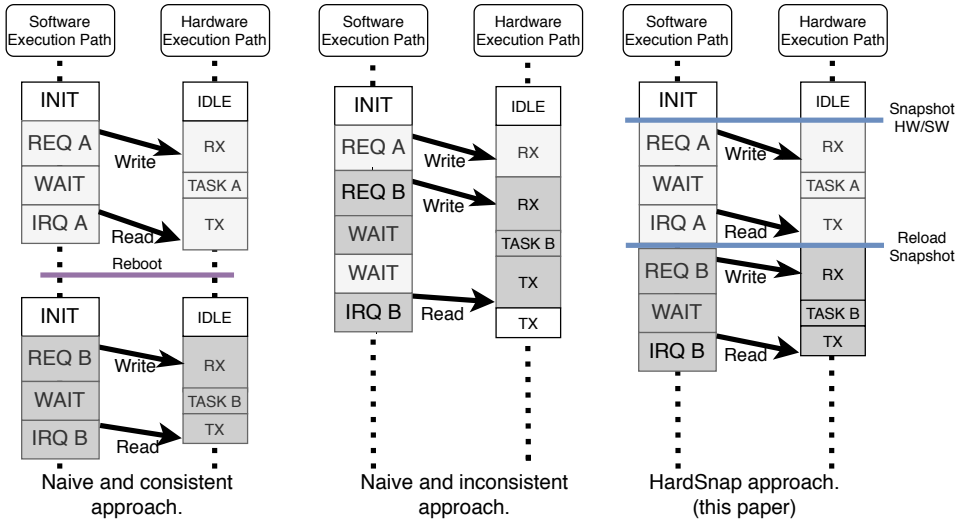


Figure 6.1: Description of different Hardware/Software co-testing execution. From left to right: naive and consistent but slow approach; naive and fast but inconsistent approach, HardSnap approach.

6.3 Design Objectives

We identify four desirable properties that the system should have.

1. Instrospective: hardware/software state introspection at any time.
2. Fast: minimize performance slowdown on the dynamic analysis tool. In particular, when saving/restoring tested system state.
3. Scalability: fit with both small and large hardware designs.
4. Automation: minimize manual intervention and automate required modification on the hardware design.

With these objectives in mind, we build HardSnap an extensible tool for system-wide testing of hardware and firmware. In particular, it aims to offer a fast and reliable way to fully control hardware state during testing. Our approach can be used to verify both firmware and hardware. Although limited by the need for hardware design sources, our solution is suitable in the context of industry or open hardware. In the following, we give architecture details regarding the implementation of main components.

6.4 Design Overview

In this section, we provide an overview of HardSnap, an advanced framework designed for security testing of hardware/software co-designed systems. In particular, it offers an efficient and consistent solution for source-based selective symbolic analysis of embedded systems. Generally, symbolic analysis leans on snapshotting mechanisms in order to finely manipulate the system under test. This is particularly relevant for testing multiple execution paths of a system at the same time, or to reduce the time spent in rebooting the system. However, this mechanism requires a full-visibility and full-controllability over the tested system. This is generally difficult to achieve with silicon-based hardware peripherals, that expose a very limited memory interface to traditional software and remote debugger.

HardSnap overcomes this problem, and it offers a selective symbolic execution engine with a high introspection level on hardware peripherals. In particular, HardSnap is built around three main components. First, a **Peripheral Snapshotting Mechanism** that takes as input a model of the hardware peripheral written in Verilog, and inserts an introspection mechanism to observe and control the internals of the peripheral. The resulting peripheral model supports snapshotting, and it can run on a simulator or an FPGA device following the design complexity and user-defined configuration. Then, a **Selective Symbolic Virtual Machine** executes the firmware programs while redirecting hardware interactions to hardware peripherals. This virtual machine is based on Inception (see Chapter 5), a framework for firmware program analysis based on the Klee [27] symbolic execution engine. We emphasize that our approach is not specific to Inception, and can be extended to any hardware-in-the-loop dynamic firmware analysis tool, such as, fuzzers, or other symbolic execution engines which requires hardware interaction. HardSnap inherits from Klee the runtime detection mechanism for memory corruptions, and it offers an interface to write assertions that are especially relevant for the detection of peripherals misuse. Furthermore, it enables security analysts to write a software-based testbench, and it generates test cases thanks to the symbolic execution engine. HardSnap enables pre-production co-testing of hardware and firmware, where both are generally designed and implemented simultaneously. For example, an embedded software developer can test hardware drivers even if the full design is not available. Finally, a **Snapshotting Controller** enables the virtual machine to generate complete snapshots of the system under test, including hardware peripherals and firmware memory. Snapshots can reduce the time to fix bugs by offering a complete view

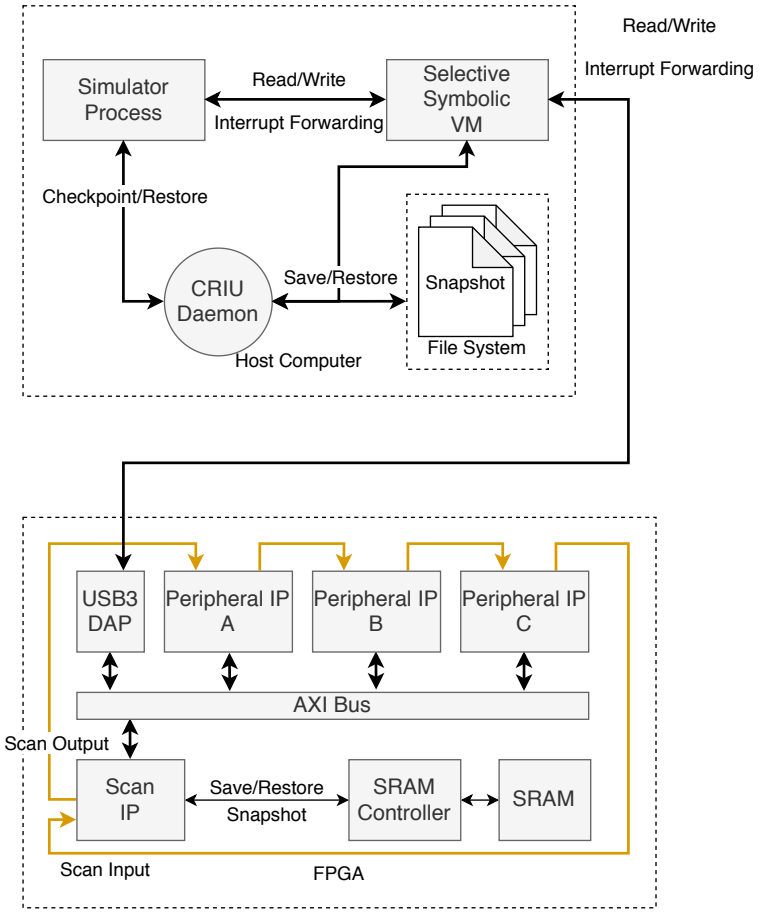


Figure 6.2: Overview of HardSnap.

of the peripheral state. To guide the reader during our explanation, we provide a description of HardSnap in Fig. 6.2.

6.4.1 Peripheral Snapshotting Mechanism

This component is the core element of HardSnap. It instruments peripherals with an introspection mechanism. The latter refers to two important notions: controllability and observability. Controllability refers to the ability of controlling the state of (all memory elements) of a peripheral at any time. Visibility refers to the ability of inspecting peripheral's state at any time. By combining both visibility and controllability, our snapshotting mechanism can inspect the internals of peripherals to save/restore peripherals' state. Generally, peripherals are modeled using a Hardware Description Language (HDL) that offers an Intermediate Representation (IR), abstracting the underlying layers (i.e., transistor or gate level). Among existing HDLs, Verilog is certainly one of the most adopted in the industry. This language adopts the register-transfer level (RTL) abstraction that describes synchronous digital circuit in term of hardware registers linked with each other through digital signals (data flow) mixed with logical operations. These hardware registers are memory elements that generally synchronize the circuit operations at each rising/falling edges of the clock signal. They directly reflect the internal state of the hardware peripheral, and they enable inferring combinatorial logic values. Our hardware snapshotting mechanism focuses on inspecting and controlling the value of these hardware registers. To support efficiently small or complex hardware design alike, we designed two different approaches to get full controllability and visibility on hardware registers.

Simulator Target. Hardware simulators are software programs able to compile and evaluate expressions written in HDL. They are good candidates for hardware snapshotting as simulated peripherals state is represented by memory variables that are easily accessible on a host computer. Furthermore, they often expose an interface to access operating system's capabilities. This is particularly interesting for attaching a remote interface (i.e., our selective symbolic virtual machine). However, simulators are extremely slow at testing complex design such as a complete System-on-Chip. To cope with design complexity, HardSnap falls back on system partitioning. In particular, it simulates peripherals only, whereas it executes firmware in a symbolic virtual machine. For this purpose, HardSnap abstracts the peripherals environment (i.e., memory bus interface) that is exposed through a remote interface to our symbolic virtual machine. In particular, HardSnap takes as input a set of Verilog-based peripherals' models and auto-

matically generates a self-contained simulator with a remote interface. This toolchain leverages Verilator, an open-source simulator, which translates Verilog-based HDL into a cycle-accurate C++-based model. The generated C++ code is then compiled and linked with HardSnap static library, which implements the remote interface and a memory bus abstraction layer that enables the remote interface to communicate with peripherals. HardSnap aims at flexibility, and it offers a modular approach where the remote interface and the memory bus abstraction can be easily replaced. We provide support for the AXI4-Lite bus interface.

Field Programmable Gate Arrays (FPGA) Target. We designed a second hardware target that focuses on performance at the cost of full execution tracing. We present this target on the bottom side of Fig. 6.2. FPGAs are post-production re-programmable integrated circuits enabling digital design emulation at a speed similar to that of the silicon chip. However, they generally offer a limited introspection and debug capability. Some FPGA manufacturers provide logical analyzers that monitor internal signals but they are very limited in the number of signals. Furthermore, these solutions are specific to the manufacturer. FPGAs generally offers limited instrospection capability like a debugger for software program. Some manufacturers offer logic readback capability to dump the FPGA fabric configuration and memory values. However, this feature is only present on a few high-end FPGAs. To avoid this limitation, HardSnap instruments the HDL of peripherals directly, so that the resulting code stays independent from the hardware target. In particular, our instrumentation toolchain takes as input Verilog-based peripheral model, and it automatically inserts a scan-chain that is basically an alternative path in which all the hardware registers form a shift register. This scan-chain is activated by a *scan_enable* signal and receives/emits input or output from a *scan_input/scan_output* signal. For completeness HardSnap also supports the readback feature of high end FPGAs and we compare the performance of readback to that of our scan-chain in Section 6.6.

6.4.2 Selective Symbolic Virtual Machine

In this context, HardSnap has been implemented on top of Inception (see Chapter 5), but with significant modifications and improvements as we will explain. In particular, we use directly from Inception its existing memory forwarding and interrupt mechanism, which enables rehosted analysis while keeping real hardware communication. Our major changes include extending the software state representation to a combined hardware/software state, a user-customizable multi-target support that routes memory

accesses to the user-selected hardware targets (i.e., FPGA or simulator), a hardware state forwarding that enables switching between hardware targets, a concretization policy that generates concrete value when a symbolic value reaches the boundary of the virtual machine domain. In addition, we enhanced Inception with engineering improvements to support recent version of Klee and to simplify future updates of Klee components.

Selective Symbolic Execution. The term selective symbolic execution has been first introduced by S2E [33]. It refers to the ability to execute symbolically the code of interest while executing concretely external resources. HardSnap symbolically executes firmware programs while executing concretely peripherals. For this purpose, it offers a concretization policy that we describe latter in this section.

Multi-target orchestration. An important improvement we made to Inception is the multi-target approach that enables user to precisely control and observe running analysis. This feature is built on top of the Inception memory forwarding mechanism. The former originally supports I/O forwarding to a unique target through Steroids. We extended this mechanism to our simulator and FPGA target. We developed custom drivers for both targets. The simulator target is remotely accessible through a shared memory. The FPGA target emulates the Inception USB 3.0 low latency debugger that we modified to receive USB 3.0 commands, and to generate AXI transactions so that it can directly access peripherals without any JTAG interface. Additionally, we created the target orchestration system. In particular, it supports state transfer from one target to another one at any time during the analysis. We believe this feature is interesting for different reasons. First, it enables to cope with targets limitations that generally offer either speed or full traces. For example, the Verilator-based target enables full visibility along the execution (i.e., traces), however, it is significantly slower than the FPGA-based target that does not offer full traces. The target orchestration enables to start the analysis on the FPGA target and once a particular point is reached the FPGA state is transferred to the Verilator target.

Concretization policy. When the symbolic domain (i.e., symbolic values) requests access to the concrete domain (i.e., hardware peripherals), our system needs to concretize the symbolic expression to a set of possible concrete values. This step is automatically done by HardSnap during symbolic execution, and it is user-customizable to choose between completeness (i.e., all possible values are tested) or performance (i.e., only one possible value is tested).

6.4.3 Snapshotting Controller

In Fig.6.2, we present a general description of HardSnap and its snapshotting controller. This controller is in charge of saving/restoring snapshots that are identified by a unique identifier. Our system supports two different hardware targets. Each target has a specific snapshotting method. The core of the snapshotting controller is part of the virtual machine and it communicates with target-specific snapshot controllers.

For the simulator target, we use CRIU a Linux userspace framework which is able to checkpoint and restore a process. Before any save/restore of the simulator process, the snapshot controller flushes all pending read/write operations, and then it freezes the simulator process. In fact, the simulator has a remote interface to send read/write commands that is an operating system capability outside the scope of the simulator. Once the simulator process has been frozen, a checkpoint is stored on a persistent storage (i.e., the file system).

On the FPGA-based hardware platform, an internal hardware block ("IP") manages hardware snapshots. This IP is driven through memory mapped registers that are directly accessible on the system memory bus, or through the USB 3.0 debugger. This IP saves and restores the peripherals state, by driving the scan-chain previously inserted. It takes as input the snapshot source address and a destination address for the scan-chain output. Once started, it suspends the hardware execution and saves all its content at the specified memory address. At the same time, it loads the specified snapshot to overwrite the hardware registers. For performance reasons, the scanning IP saves peripherals snapshots in an SRAM memory. This optimization significantly reduces the time taken for saving or restoring hardware peripheral state.

6.5 Architecture and Implementation

In the following, we describe the details of our system architecture and implementation. We first describe hardware snapshotting and then our symbolic virtual machine.

6.5.1 Hardware Snapshotting Instrumentation

To support small and complex hardware designs alike, we use two approaches (Fig. 6.3): purely software simulation or FPGA backed simulation.

Simpler hardware components can be simulated purely by software. For this, we extend `Verilator` [88], an open-source Verilog simulator designed

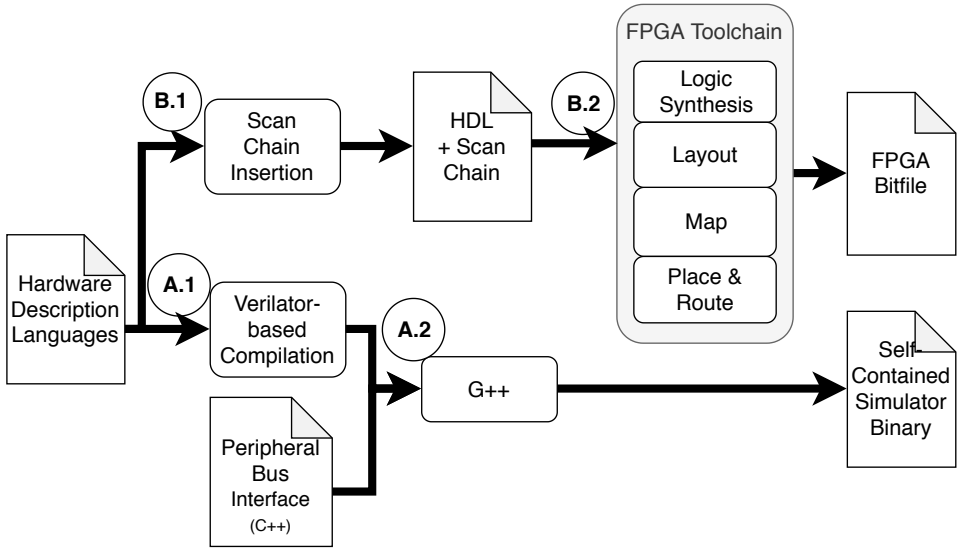


Figure 6.3: HardSnap's instrumentation toolchain.

with performance in mind. Verilator transforms a hardware component written in the Verilog Hardware Description Language into a self-contained multithreaded C++-based simulator (A.1). HardSnap automatically extends this simulator with a remote interface to connect the simulated hardware to an external client. This interconnects a simulated memory bus (i.e., AXI, Wishbone) to a remote communication interface (i.e., socket, shared memory). With such an interface an external application, such as our symbolic virtual machine, can reach the peripherals (i.e., memory mapped registers). The resulting C++ code is then compiled using `g++` to generate a self-contained simulator program (A.2). This solution is suitable for testing relatively simple hardware designs, but is too slow for complex peripherals.

To test more complex designs, HardSnap emulates the hardware block on an FPGA. This approach scales well for complex designs, as long as the component fits in the FPGA, but it does not offer any visibility on the internals. For this reason, we built a tool which instruments HDL files to insert a scan-chain, which provides access to all memory elements of the design (memories, registers, etc). Knowing the value of hardware registers, enables us to infer the value of combinatorial elements. This

instrumentation is done directly at the RTL level (i.e., Verilog) (B.1), the instrumentation is therefore independent from the FPGA toolchain. User-defined parameters allow to limit the instrumentation to a sub-component of the entire design. Finally, the normal FPGA toolchain is used to generate a bitfile describing the configuration of the FPGA fabric (B.2).

6.5.2 Selective Symbolic Virtual Machine

We extended INCEPTION's symbolic virtual machine state representation from software only to also consider hardware state. We first define the notion of software state and hardware state:

- **Software State**: A software state, under KLEE, is a 3-tuple S^{sw} {PC,F,G} of a program **P** at a time **t**, where PC is the program counter, F is a set of Stack Frames (i.e., local variables) and G is the global memory (global variables and heap).
- **Hardware State** A hardware state, under HARDSNAP, is a set S^{hw} of all the hardware registers values of the hardware peripherals under test at a time t. We refer to this as a snapshot when it is an offline representation and refer to target when designing the hardware platform.

Update of the state representation in INCEPTION (from KLEE) is straightforward. Each software state S^{sw} is associated to a unique hardware snapshot identifier. Thereafter, we refer to S which includes S^{sw} and S^{hw}. Algorithm 1 describes the main execution loop algorithm of our modified version of INCEPTION. A set **AS** contains the active states and is initialized with the initial state (PC at program entry point and stack empty, no corresponding hardware snapshot). A variable $S_{previous}$ keeps reference of the previous state that is being processed and is initialized as empty. Then, the main process iterates until **AS** becomes empty, i.e., there is no more state to test. This process is as follows.

First, a call to `SelectNextState` returns the next state to evaluate, with respect to the user-defined state selection heuristic. This is the original behavior of KLEE, that has been extended by INCEPTION to avoid selecting a different state if the previous one is processing an interrupt. This mechanism makes interrupt atomic to reduce timing violations. Then, we added a mechanism to detect modifications on **S** by comparing its ID with $S_{previous}$ ID's. When the comparison fails, it indicates that current hardware state does not belong to current software execution.

We build two mechanisms to manipulate hardware state on demand. First the function `UpdateState`: suspends hardware target, generates a new snapshot and finally resumes the target execution. The new snapshot overrides the snapshot associated with $S_{previous}$. Then, `RestoreState` overrides the current hardware state with the snapshot associated with **S**. Doing so, we ensure that further interactions will only affect the corresponding state. This hardware context switch is a crucial mechanism to guarantee that testing software state interacts with the correct hardware state. The same mechanism is applied when the symbolic machine forks software state (e.g., symbolic condition on a branch). In this case, resulting state flows with a unique and non-shared hardware snapshot.

Algorithm 1: Pseudocode of HardSnap's main execution loop.

```
1  AS = {S_init};
2  S_previous = ∅;
3  while AS ≠ ∅ do
4  |  S = SelectNextState(AS,S_previous);
5  |  if S_previous ≠ ∅ and S ≠ S_previous then
6  |  |  UpdateState(S_previous );
7  |  |  RestoreState(S);
8  |  end
9  |  S_previous = S;
10 |  ServePendingInterrupt(S);
11 |  StepInstruction(S);
12 |  S_new = ExecuteInstruction(S);
13 |  AS ⇐ AS ∪ S_new;
14 end
```

6.6 Evaluation

In the previous part, we have described how we implemented hardware shapshotting on top of Inception. In this section, we undertake experiments on a corpus of 4 synthetic real world and open-source peripherals. We selected these peripherals because they are common on embedded systems and have different design complexities. We made three experiments on these peripherals. With these experiments, we seek to answer three questions.

How long does it take to save/restore a hardware state? In order to answer this question, we measured the saving/restoring process duration for our corpus of peripherals on each proposed hardware snapshotting methods (i.e., simulator, FPGA with scan-chain, and the readback

feature that is manufacturer dependant). For each of them, we compared the hardware design size with the duration to determine how it may impact performance. In addition, we complete the performance evaluation by measuring the I/O forwarding latency and execution speed between the FPGA and the simulator target.

How beneficial is hardware snapshotting for firmware analysis? For this purpose, we measured the execution speed and the analysis consistency that are two crucial factors for dynamic firmware analysis. We run these experiments on HardSnap and Inception. First, the execution speed increases the number of test cases the system can evaluate per unit of time. This increases the probability of discovering bugs. Second, we demonstrate how corrupted hardware states affect the analysis accuracy. This may increase the number of false negatives or false positives. For example, a firmware executing an interrupt handler while the peripheral is not active. Using this experiment, we show how hardware interactions may affect the accuracy of the firmware analysis and at the same time we evaluate the correctness of our approach.

How usefull is HardSnap for hardware testing? We present a case study to demonstrate the versatility of HardSnap which can be used for hardware and software co-testing.

All experiments were run on Ubuntu 18.04 (Linux kernel 4.15.0-42-generic) with an Intel core I5 4500U 3.00GHz and 12GB RAM. All the presented experiments are based on a corpus of 4 hardware peripherals presented below.

- *SHA256 peripheral* [89] is a Verilog-based implementation of a standard cryptographic hash function with a wrapper to interface it with a memory bus (i.e., AXI-Lite Slave). The peripheral is part of the Cryptech open HSM platform [8] that is deployed on commercial HSM.
- *AES Counter Mode* [54]. This IP enables encryption/decryption using the AES in CTR mode. It is commonly used in wireless communication protocol such as WPA2 for WiFi.
- P*rogrammable Interrupt Controller* (PIC) is a software programmable interrupt controller. Since firmware programs are generally interrupt-driven, this peripheral is extremely important for firmware analysis.
- *TIMER* peripheral is a simple Verilog-based timer with status and control registers. Firmware can configure the interrupt timer frequency, turn it on/off.

We used two FPGA development boards for our experiments: A ZedBoard with a Zynq-7000 ARM/FPGA SoC and an Ultra96 Zynq UltraScale+ ZU3EG development board. The ZedBoard is used for all experiments except for measuring the performance of the readback command which is only supported by the UltraScale+ board.

6.6.1 Experiment I: Hardware Snapshotting Performance

For our first experiment, we measure hardware snapshots performance for each proposed hardware snapshotting methods.

Experiments Details Each experiment consists of the following: first, the hardware target is started (i.e., FPGA, Verilator-based simulator), and a control program is started (CRIU service runs as a Linux daemon in background). This program is a C++ based application which drives the save/restore process and measure time per operation. It is able to save and restore the hardware state for any supported platforms. It commands CRIU services through a socket to deal with the simulator platform and it communicates with the USB3-based Inception-debugger to save and restore an FPGA snapshot. The program measures the time to save the current peripheral state and time to restore previously saved state. We repeat this step 10^6 times for each peripheral and report the average time and standard derivation in Figure 6.4. Additionally to this experiment, we provide in Table 6.2 information regarding the size of the design under test that we measure in terms of the number of Flip Flops (i.e., the scan-chain length), simulator binary size, and bitfile size. The latter is relevant to the evaluation of the readback feature that dumps all the FPGA configuration and its memory values. This operation generates a bitfile that contains the whole FPGA configuration, including for the unused FPGA logic. Therefore, the bitfile size depends on the FPGA model and not directly on the complexity of the tested design.

Design	Number of Flip Flops	Simulator Size	Bitfile Size
ALL[1]	10817	7986 kB	5568 kB
AES CTR	9712	1541 kB	5568 kB
SHA 256	999	1209 kB	5568 kB
PIC	41	1189 kB	5568 kB
TIMER	65	1185 kB	5568 kB

Table 6.2: Size of the test design corpus.

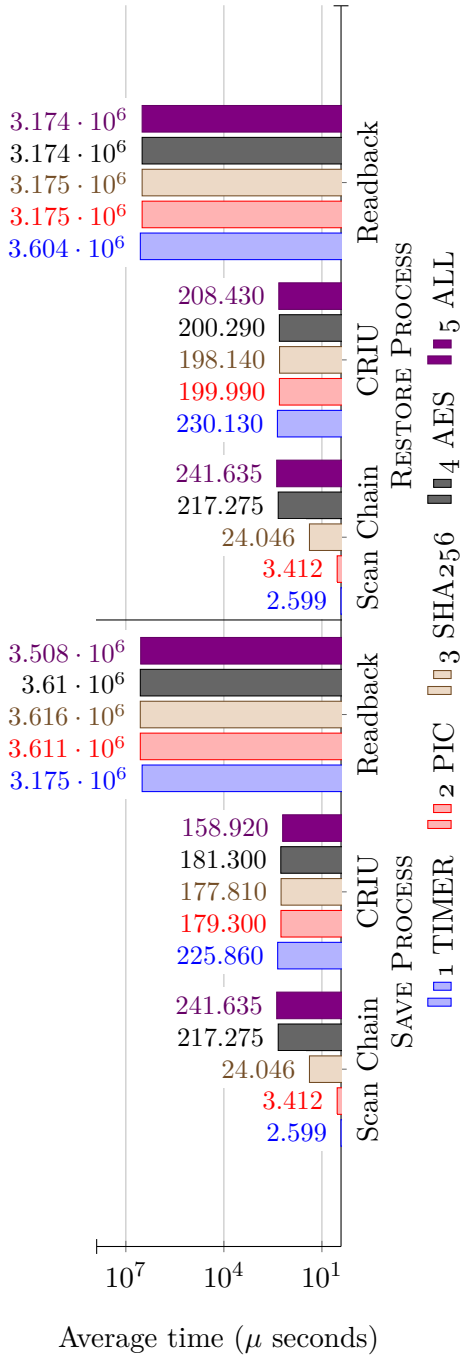


Figure 6.4: Average duration, in microseconds, for 10^6 snapshot saves or restores for the FPGA and the simulator. Note the y-axis is plotted on a logarithmic scale.

Observations and results By saving/restoring the hardware state, we observe that the hardware is still responding between each test, indicating that this process works correctly, even during stress tests.

We can first observe on Figure 6.4 the readback method does not perform well. In fact, it collects more information than needed as all the FPGA fabric configuration and memory values have to be accessed. According to the FPGA documentation, the number of hardware registers (Data Flip Flop) is 1.41×10^5. Extrapolating the results, we find that our scan-chain method would remain 5 times faster than the FPGA readback.

We also see that, for our corpus, the scan-chain method is the fastest on average. It is faster or has comparable performance than the CRIU simulator snapshotting. At first glance this is surprising because data transfers using the scan-chain (5MB/s) is much slower than CRIU snapshot (7.5GB/s). However, the scan-chain snapshots strictly the necessary information (design state) while CRIU snapshots the whole simulator process. This makes the software-based snapshot 738 times larger than the scan-chain based snapshot (for the larger example `ALL`). This also explains why snapshot time for with CRIU seems to be independent of design size.

Our scan-chain can also store snapshots in the FPGA's internal SRAM, without involving any software. Another optimisation we implemented is to simultaneously save and restore the hardware state, scanning in the state to restore while the state to save is scanned out. With those optimisations, in our experiments, the FPGA-based scan-chain is faster than snapshotting the simulator process when the number of hardware registers (i.e., D Flip Flop) does not exceed $9,712$ (i.e., AES size). In fact, the simulated design also has a scan chain that is used when forwarding state to/from the FPGA target. This could significantly reduce the duration time for restoring/saving the simulated peripherals, even if current results are perfectly acceptable.

Additionally to this experiment, we measure IO forwarding latency and the execution speed of our hardware targets. For the forwarding latency, we measure the average duration time to read/write mapped registers and repeat this operation 10^6 times. Using the USB 3.0 DAP, the reads take 80.36 ms while writes take 40.07 ms. Respectively, the simulator target takes 0.19 and 0.17 ms. The duration time to perform 10^6 reads/write requests for the simulator platform is 2 order of magnitude shorter than the duration time for the same action on the FPGA device. This experiment highlights the time penalty when communicating with external device. Obviously, the operating speed of the FPGA device is significantly faster than running a design in a simulator. For this reason, we complete our exper-

[1]Synthetic digital design composed of all the peripheral corpus.

iment by measuring the execution speed on both hardware targets (e.g., FPGA and Verilator). We measured the duration time to compute 1×10^6 sha256 hash. The fpga target returned in 1.088μs while the simulator targets ruturned in 30 ms. Our multi-target approach enables user to balance between performance following the design complexity.

To conclude, HardSnap supports different snapshotting methods where performance range from 1.5MB/s to 7.5GB/s and where the duration time to restore/save any peripheral of our corpus does not exceed 240 μs. This experiment highlights the improvments that HardSnap offers for hardware-in-the-loop analysis.

6.6.2 Experiment II: Gain for Firmware Analysis Tools

In our second experiment, we seek to evaluate the benefit of using hardware snapshotting on firmware analysis. In particular, we focus on measuring the execution speed and the analysis consistency.

Experimental Details For the purpose of this evaluation, we created a program generator that given a program complexity 'N' generates a code composed of N levels of nested branches where each branch condition depends on a program input value. This value is the operating mode that indicates which hardware components is used (i.e., AES or SHA256). In addition, we generate random operations at each branch to prevent compiler optimizations, and to randomly change the value of the operating mode. Those operations are randomnly selected between AES and SHA256 computations. Both operations rely on the corresponding hardware accelerator, and therefore accesses are redirected to the hardware target.

In order to detect inconsistencies during analysis, we added assertions in the code. In particular, we detect incorrect IRQs and incorrect hardware outputs. Incorrect IRQs are detected using a token mechanism that counts the number of AES/SHA operations that we compare with the number of executed interrupt handlers. The difference is the number of interruptions that have been (incorrectly) not executed. In addition, we add assertions to verify that each interrupt request belongs to the correct execution path and hardware peripheral. Incorrect hardware outputs are detected by comparing outputs with expected values. We run experiments with 'N' ranging from 0 to 10 for Inception (no hardware snapshot) and HardSnap with three different Klee state selection heuristics: Min-Distance-to-Uncovered (MD2U), Depth-First Search (DFS), Random Path (RP). We present the cumulative results in Figure 6.5.

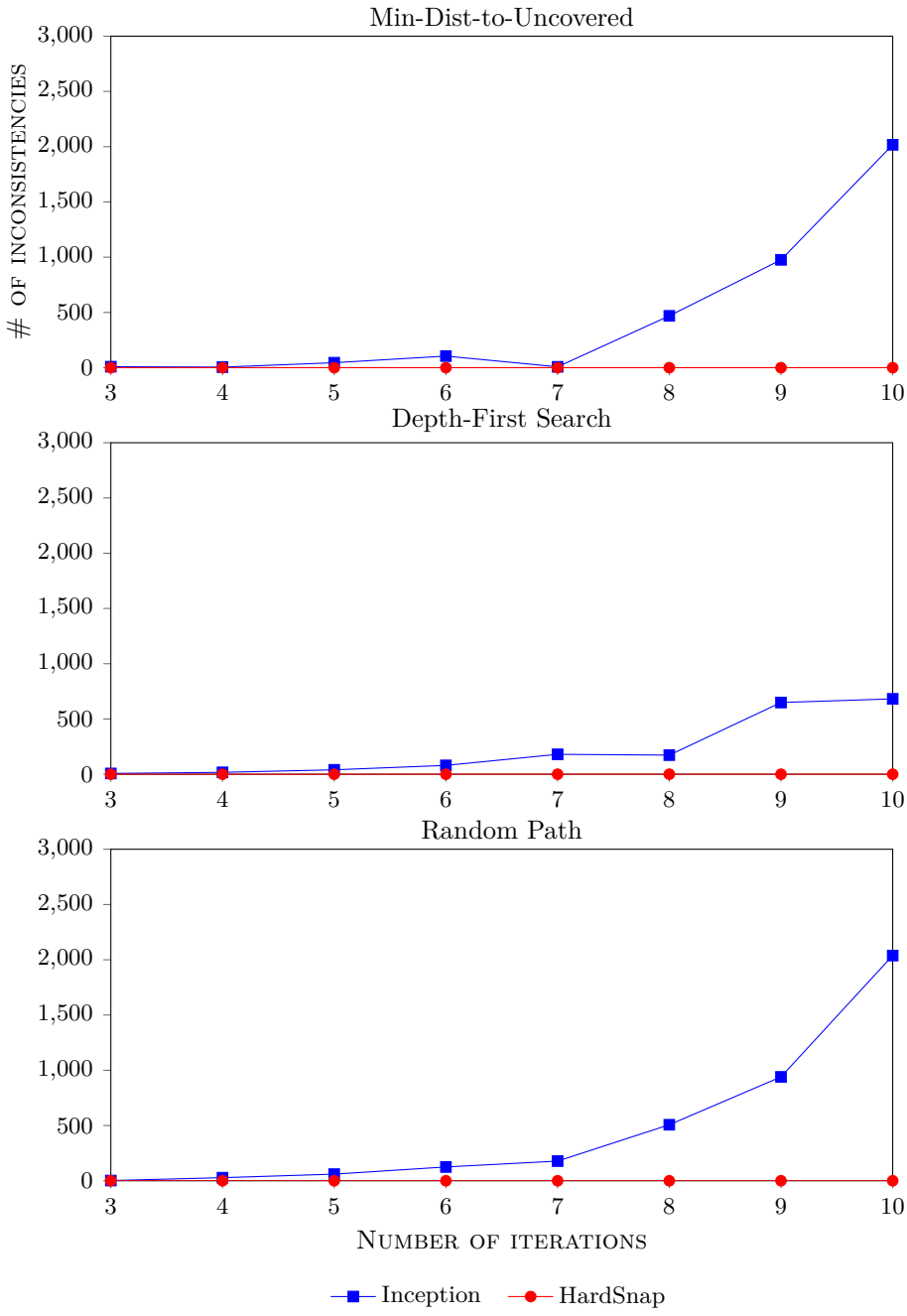


Figure 6.5: The cumulative number of inconsistencies found in a synthetic firmware with different state selection heuristics by the number of iterations.

Then, we run the same experiment but this time we increase the number of interactions to 10^6. This value is inspired by the number of interactions reported by Talebi et al. [93]. For this part of the experiment, we modify Inception to use a record-and-replay approach for restoring hardware states when switching branches. In fact, in Inception, no synchronization mechanisms are used to avoid inconsistent state during testing (see Chapter 5). Instead of restarting the execution from zero (which is extremely slow), we choose to implement the record-and-replay mechanism from Avatar [98]. We report the results in Figure 6.6.

Observations and results First, we present the results for the consistency analysis. Our results show inconsistencies only when our hardware snapshotting mechanism is not enabled (Inception). This demonstrates the capability of HardSnap. Contrarily, Inception obtains an important and increasing number of inconsistencies for all the tested search heuristics. The number of inconsistencies in the worst case are 2038 for Random-Path, 682 for DFS and 2017 for MD2U. It is notable that the Depth-First Search (DFS) presents less inconsistencies than the two others search heuristics. This is consistent with the fact that DFS only change execution path when the current one returns. Thus limiting the number of context switches, and therefore the number of inconsistencies. Those inconsistencies are important as they can lead to false positives or false negatives. Furthermore, they would require significant work for an analyst to understand and filter them.

Second, we present results for the execution speed measurements. Our results show an average performance enhancement of 3.34 for HardSnap over the record-and-replay approach. Moreover, when N=9 (i.e., 512 explored states), HardSnap is 8.9 times faster, and when N=10 Inception does not complete after running for 24 hours while HardSnap finishes in roughly 2 hours.

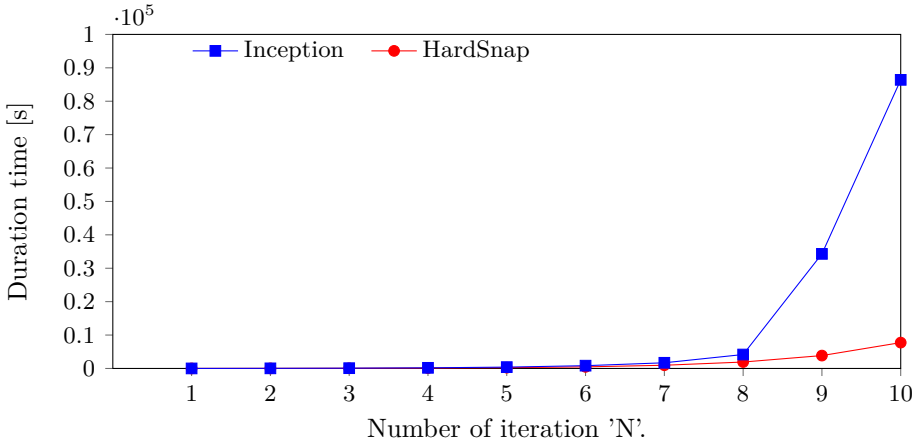


Figure 6.6: The duration time by the number of iterations 'N' (Number of explored states=2^N).

6.6.3 Experiment III: Case Study

While so far we looked at how HardSnap improves firmware testing, this case study demonstrates how HardSnap can be used to test the hardware. For this purpose, we inject a synthetic bug in the previously described PIC. This synthetic bug corrupts the current interrupt ID. The PIC has two operating modes: priority and polling mode. In this example, we aim at verifying the correctness of the priority mode, which selects the active interrupt based on configurable interrupts priorities. For this purpose, we wrote a small testing code (Figure 6.7). This is a good example of a difficult bug to discover as it requires exploring several logical states to be triggered.

Using this test with HardSnap generates 343 test cases and in 67.61 seconds (5 test cases per second; 21432 instructions in total). In total 127 assertions failed and, for each the state is transferred to the simulator. The simulator can produce a waveform, which gives a good view of the internal hardware state.

This simple case study shows two important features of HardSnap. First, the symbolic execution provides a large coverage with just one test case (i.e., the combinations of interrupt priorities). This significantly reduces the workload when compared to manually writing each test. Second, when a violation is found on the high performance FPGA, the multi-target approach allows to transfer a state from hardware to the simulator, with much better visibility. This makes debugging easier.

```
// the PIC supports 7 interrupt sources [1:7]
uint8_t i1, i2, i3 = klee_int();
klee_assume(i1>=1&i1<8&i2>=1&i2<8&i3>=1&i3<8);

// set the interrupt priority table
// right to left : highest to lowest priority
pic[1] = (i3<<6) | (i2<<3) | i1; pic[0] = 2;

// trig interrupt i1, i2, and i3
pic[2] = (1 << i1) | (1 << i2) | (1 << i3); pic[2] = 0;

// wait interrupt signal or timeout
while( (pic[0] & 0xFF) == 0);

// check if i1 is the active interrupt
if( i1 != (pic[0]&0x7)) {
 klee_report_error(___FILE___, ___LINE___, "bug", "hardsnap");
 transfer_state_to_target("simulator");
}
```

Figure 6.7: Use case written in C to verify the correctness of the PIC peripheral.

6.7 Limitations

Limitations of FPGA-based Emulation. FPGA-based emulation has limitations. First, this technique focuses on emulating digital functions rather than analog functions. This makes the emulation of some components difficult. Second, the scan-chain may impose strong constraints for the synthesizer, and it may require a slowdown of the nominal frequency. While ASIC-based design generally relies on specific scan Flip-Flop to form the scan-chain, such blocks are not common on FPGA, making it less efficient.

Asynchronous Logic. The design under test may interact with a circuitry that cannot be instrumented. For instance, our USB 3.0 interface is asynchronous, and cannot be fully controlled by HardSnap. This may lead to inconsistent state (i.e., interrupt mismatch). To overcome this issue, we made two modifications. First, we added a hardware register in the scan-chain to store the ID of the current executions state. Then, we forward this ID in addition to the interrupt request.

6.8 Conclusion

In this chapter, we introduced the concept of *hardware snapshot* to improve hardware/software co-testing. We demonstrated how HardSnap improves system-wide analysis with a high visibility over the overall system,

enabling hardware introspection at any time during the analysis. The results of our experiments show that HardSnap improves both hardware and firmware analysis. It significantly reduces the bottleneck or inaccuracy with hardware-in-the-loop approaches. We also demonstrate that inconsistencies may affect the analysis when naively testing firmware programs. These inconsistencies may affect the analysis correctness, and they may lead to false positives or false negatives. With HardSnap frequent reboots and replay are not needed anymore. HardSnap is open-sourced to make our results easily reproducible, and is available at https://github.com/hardsnap/.

Chapter 7

Conclusion and Future Work

In this chapter we conclude the thesis. Then, we discuss possible research directions and future work.

7.1 Conclusion

In this thesis, we explore the problem of embedded systems security analysis and consider the point of view of a security analyst having access to the source code and to Hardware Description Languages (HDL). For this purpose, we consider programs and hardware peripherals issued from the real-world.

To begin with, in Chapter 3, we study the security analysis of a complex system-on-chip. For this purpose, we propose a naive methodology for testing such systems. This methodology has proved its efficiency during a hardware bug finding competition (HackDack 2019). From this experience, we outline the research directions and challenges that we tackle in the rest of this thesis. In particular, we outline three main limiting factors: (1) partial-emulation latency, (2) semantic differences, (3) limited visibility and control on the system under test. We designed and developed three independent solutions that we evaluated with real-world systems.

First, Steroids makes partial-emulation testing practical by reducing the I/O latency between the dynamic analysis tools and the system under test (i.e., embedded systems). In particular, Steroids offers mechanisms to redirect interactions (i.e., mapped memory, DMA, interrupt) between the real device and the remote analysis tool.

Second, Inception enables system-wide analysis of firmware programs. Previous work was either testing binary or source-code, whereas Inception supports hybrid analysis where source-code may contain assembly lines and even binary dependencies. Inception tool has been used in an industrial production flow where it detected memory corruptions on bootloader stored in mask ROM. In addition, we evaluated the correctness of our approach through 5.2×10^4 test cases that compare the operational semantic in Inception with the real hardware. Furthermore, we evaluate the fault detection of Inception by testing 1.5×10^3 test case from the NIST test suite. Finally, we demonstrate how efficient source-based analysis is compared to binary-based analysis and how our hybrid approach may improve fault detection in binary code.

Third, we tackle the complex problem of limited visibility and control when testing embedded system. These two notions are crucial for detecting bugs, analysis performance and consistency.

To conclude, we believe that this thesis offers considerable improvements

to the methods used for analyzing the security of embedded systems in pre-production.

7.2 Future Work

This thesis presents fundamental improvements for achieving system-wide source-based security analysis of embedded software. However, despite the considerable undertaken work in this thesis, there are several lines of research arising from this work and potential improvements.

First and foremost, omitting large comparisons with related work arise questions on the efficiency of our approach. However, due to the diversity of embedded systems architectures, it is often impossible to test embedded software on all the existing security analysis tools without important modifications. For this reason, we focus our comparison with related work that supports the same architecture. To facilitate replicability of our work for future research [4], we provide detailed information regarding the set-up; we based our material on off-the-shelf components; we provide all the source code and test cases [7]. We would like to emphasize the fact that test suites are common for desktop applications, however, as far as we know there are no such tests for evaluating embedded systems security analysis tools.

Second, Inception has been actively used to test industrial applications under development (i.e., bootloader) where it shows interesting results. In Chapter 5, we present an evaluation of different publicly available firmware. Among them, embedded operating systems (OS) are pervasive today. There is an increasing number of OSs and therefore we thought that evaluating their security with Inception would be extremely interesting.

In this thesis, we introduced the notion of hybrid analysis of source-code and binary code. Our implementation, detailed in Chapter 5, presents a hand-written lifter that translates binary code to Intermediate Representation (IR). Writing a lifter is error-prone and difficult as it requires a deep understanding of the Instruction Set Architecture (ISA) for which the specification may contain errors. Errors in the lifted IR can affect the analysis accuracy (i.e., false-positive or false-negative). For this reason, we decided to compare thoroughly each emulated instruction with a real implementation (i.e., a silicon chip). However, we believe that lifter could be automatically generated from the ISA architecture and verified using semantic equivalence techniques. This approach would also improve architecture support.

Finally, this thesis aims at analyzing system-wide embedded systems. Our solution Inception focuses on firmware analysis while our approach HardSnap extends Inception to hardware and software co-testing. We

believe that research directions arise from this work. First, symbolic execution, which is widely used for testing software, could be applied to verify hardware peripherals. Second, analyzing analog components would enable a fine-restriction of the symbolic domains to reduce analysis complexity.

Appendices

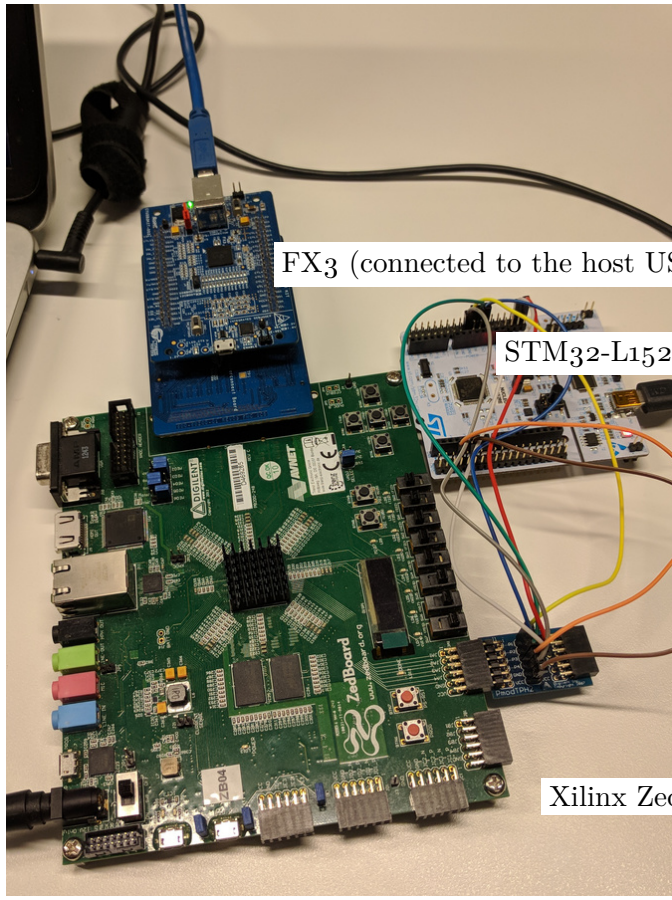


Figure 1: Hardware components of the Inception system using an STM32 demo board using an Arm Cortex-M3.

Figure 2: Example program with mixed source and assembly. (1) the original C source code with inline assembly code. (2) CLang generated LLVM bitcode. (3) mixed-IR: LLVM bitcode with produced by merging lifted bitcode with CLang generated bitcode. We use the naked keyword to limit the size of the example.

.1 Examples of IR level adaptation

1. **High-IR to low-IR parameters passing.**

```
define i32 @foo(i32 %a, i32 %b) #0 {
entry: // PROLOGUE BB
  store i32 %a, i32* @R0
  store i32 %b, i32* @R1
  br label %"i32x4_reti32+0"

"i32x4_reti32+0":
  ...
  //EPILOGUE
  %0 = load i32* @R0
  ret i32 %0
}
```

2. **Low-IR to high-IR parameter passing.**

```
void @high_function(){
... // High IR code

%R0_2 = load i32* @R0
%R1_1 = load i32* @R1
%R2_1 = load i32* @R2
%R3_2 = load i32* @R3
%SP15 = load i32* @SP
%SP16 = inttoptr i32 %SP15 to i32*
%SP17 = load i32* %SP16

%0 = call i32 @low_function(
       i32 %R0_2,
       i32 %R1_1,
       i32 %R2_1,
       i32 %R3_2,
       i32 %SP17)

store i32 %0, i32* @R0

... // High IR code
}

define i32 @foo(i32 %a, i32 %b,
  i32 %c, i32 %d, i32 %e) #0 {
  ... // low-IR
}
```

Type	Board	Sample(s)	Number	Generation	Golden Model	Automated Functionality Check	Stable
Forwarding hardware	None	Test-bench	1	Manual	Python model	✓	✓
Forwarding driver	Any	IO benchmark	1	Random, manual	Property	✓	✓
Single instructions	Any	Translator-verif	50k	Random	Native regs/stack	✓	✓
Sequences, control flow	Any	Translator-verif	3k	Random	Native regs/stack	✓	✓
Feature-specific	Any	Inception-samples	13	Manual	Property	✓	✓
Simple algorithms	Any	Inception-samples	9	Manual	C version	✓	✓
Complex algorithms	STM32L152RE	Arm DSP library	4	Collected	Hardwired result	✓	✓
Complex features	Host only	mini-arm-os	3	Collected, manual	Behavior	✗	✓
Important KLEE regressions	Any	Examples	102	Collected	Property	✓	✓
Dhrystone v2.1	Host only	Performance benchmark	1	Collected	Property	✓	✓
NIST Klocwork based	Any	Vulnerable examples	40	Collected	Property	✓	✓
expat based	Any	Vulnerable examples	16	Collected	Property	✓	✓
Interrupts and multithreading	STM32L152RE	Vulnerable examples	5	Manual	Property	✓	✓
Simple demos	LPC1850DB1	Drivers for LEDs, buttons, ADC, Ethernet, Web server	5	Collected	Native behavior	✗	✓
	STM32L152RE	Drivers for LEDs, buttons, UART (ST and libopencm3)	5				✓
		Temperature via UART (libopencm3)	1				✓
	(Anonymized)	Drivers for LEDs	1				✓
Complex demos	STM32L152RE	FreeRTOS 2 threads	1	Collected	Native behavior	✗	✓
		ChibiOS	1				✗
		MbedOS	1				✗
	(Anonymized)	Bootloader	1				✓
		SDK	1				✓
		Smart Card Reader	1				✓
		MbedTLS 2.6.0	1			✓	✓
FreeRTOS and NIST Juliet	STM32L152RE	Vulnerable examples	1562	Collected			✓

Table 1: Summary of validation tests and results.

References

[1] https://debian.org.

[2] https://getfedora.org/.

[3] https://archlinux.fr.

[4] Artifact review and badging.

[5] Dstream: Debug and trace.

[6] Flyswatter. https://www.tincantools.com/flyswatter/.

[7] The inception framework.

[8] Making the internet a little bit safer. *Cryptech*.

[9] Modelsim. *URL: https://www.intel.com/*.

[10] CVE-2018-16522. Available from MITRE, CVE-ID CVE-2018-16522., 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16522.

[11] CVE-2018-16525. Available from MITRE, CVE-ID CVE-2018-16525., 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16525.

[12] CVE-2018-16526. Available from MITRE, CVE-ID CVE-2018-16526., 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16526.

[13] CVE-2018-16528. Available from MITRE, CVE-ID CVE-2018-16528., 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16528.

[14] P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, Aug. 2020. USENIX Association.

[15] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *European conference on object-oriented programming*, pages 142–166. Springer, 1996.

[16] ARM. *APCS: ARM Procedure Call Standard for the ARM Architecture*, November 2015. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf.

[17] A. R. Artem Dinaburg. McSema: Static Translation of X86 Instructions to LLVM, 2014.

[18] C. S. L. at UC Santa Barbara. *angr.io*.

[19] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 2018.

[20] A. Barisani. https://github.com/f-secure-foundry/tamago.

[21] R. Barry et al. Freertos. *Internet, Oct*, 2008.

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

[23] R. Benadjila, A. Michelizza, M. Renard, P. Thierry, and P. Trebuchet. Wookey: designing a trusted and efficient usb device. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 673–686, 2019.

[24] BlackBerry. https://blackberry.qnx.com/en/products/neutrino-rtos/index.

[25] A. Bougach, G. Aubey, P. Collet, T. Coudray, J. Salwan, and A. de la Vieuvi. Dagger: Decompiling Software Through LLVM, 2013.

[26] B. Buyukkurt and I. Baev. Google groups LLVMdev RFC: Indirect Call Promotion LLVM Pass. RFC, https://groups.google.com/forum/#!topic/llvm-dev/_1kughXhjIY.

[27] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[28] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. Exe: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*. Citeseer, 2006.

[29] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[30] B. Chen, K. Cong, Z. Yang, Q. Wang, J. Wang, L. Lei, and F. Xie. End-to-End Concolic Testing for Hardware/Software Co-Validation. In *IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019.

[31] D. D. Chen, M. Woo, D. Brumley, and M. Egele. Towards automated dynamic analysis for linux-based embedded firmware. pages 1–16, 2016.

[32] M. Chiang, T. Yeh, and G. Tseng. A qemu and systemc-based cycle-accurate iss for performance estimation on soc development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):593–606, April 2011.

[33] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E Platform. *ACM Transactions on Computer Systems*, 2012.

[34] I. Chun and C. Lim. Es-debugger: the flexible embedded system debugger based on jtag technology. In *The 7th International Conference on Advanced Communication Technology, 2005, ICACT 2005.*, volume 2, pages 900–903. IEEE, 2005.

[35] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer. Halucinator: Firmware re-hosting through abstraction layer emulation.

[36] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti. PIE: Parser identification in embedded systems. In

Proceedings of the 31st Annual Computer Security Applications Conference, ACSAC 2015, pages 251–260, New York, NY, USA, 2015. ACM.

[37] N. Corteggiani, G. Camurati, and A. Francillon. Inception: System-wide security testing of real-world embedded systems software. In *27th USENIX Security Symposium (USENIX Security)*, 2018.

[38] A. Cui and S. J. Stolfo. Defending embedded systems with software symbiotes. In *International Workshop on Recent Advances in Intrusion Detection*, pages 358–377. Springer, 2011.

[39] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, pages 463–478, 2013.

[40] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *22nd USENIX Security Symposium (USENIX Security)*, 2013.

[41] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran. HardFails: Insights into Software-Exploitable Hardware Bugs. In *28th USENIX Security Symposium (USENIX Security)*, 2019.

[42] F. Digilent's ZedBoard Zynq. Dev. board documentation. *Google Scholar*.

[43] A. A. Donovan and B. W. Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[44] A. Egners, B. Marschollek, and U. Meyer. Hackers in your pocket: A survey of smartphone security across platforms. Technical report RWTH Aachen , ISSN 0935–3232, May 2012.

[45] A. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings of the First International Conference on Interactive Theorem Proving*, ITP'10, pages 243–258, Berlin, Heidelberg, 2010. Springer-Verlag.

[46] J. Ganssle. Beginner's corner - in-circuit-emulators. http://www.ganssle.com/articles/BegincornerICE.htm. Accessed: 2010-09-30.

[47] F. Ghenassia et al. *Transaction-level modeling with SystemC*, volume 2. Springer, 2005.

[48] L. Goasduff. Gartner says 5.8 billion enterprise and automotive iot endpoints will be in use in 2020. https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io.

[49] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 135–150, Chaoyang District, Beijing, Sept. 2019. USENIX Association.

[50] N. Hasabnis and R. Sekar. Lifting assembly to intermediate representation: A novel approach leveraging compilers. *SIGOPS Oper. Syst. Rev.*, 50(2):311–324, Mar. 2016.

[51] B. Hay and K. Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, Apr. 2008.

[52] G. Hernandez, F. Fowze, D. J. Tang, T. Yavuz, P. Traynor, and K. R. Butler. Toward Automated Firmware Analysis in the IoT Era. *IEEE Security & Privacy*, 2019.

[53] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2245–2262. ACM, 2017.

[54] H. Hsing. Advanced encryption standard fpga implementation.

[55] C.-Y. R. Huang, Y.-F. Yin, C.-J. Hsu, T. B. Huang, and T.-M. Chang. SoC HW/SW Verification and Validation. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2011.

[56] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[57] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[58] K. Koscher, T. Kohno, and D. Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *WOOT*, 2015.

[59] C. S. D. Laboratory. Fracture: architecture-independent decompiler to LLVM IR, 2013.

[60] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[61] D. X. Li, R. Ashok, and R. Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 53–61, New York, NY, USA, 2010. ACM.

[62] H. Li, D. Tong, K. Huang, and X. Cheng. FEMU: A firmware-based emulation framework for SoC verification. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, 2010.

[63] I. B. Mahapatra, S. Natarajan, Nalesh S, and S. K. Nandy. SIMAAH: RTL simulation accelerator for complex SoC's. In *IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2015.

[64] D. Maier, B. Radtke, and B. Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, Aug. 2019. USENIX Association.

[65] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 261–272, New York, NY, USA, 2009. ACM.

[66] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[67] MITRE. 2019 cwe top 25 most dangerous software errors, 2019.

[68] S. Mrdovic, A. Huseinovic, and E. Zajko. Combining static and live digital forensic analysis in virtual environment. In *2009 XXII International Symposium on Information, Communication and Automation Technologies*, pages 1–6. IEEE, 2009.

[69] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti. Avatar²: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research (colocated with NDSS Symposium)*, BAR 18, February 2018.

[70] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, San Diego, UNITED STATES, 02 2018.

[71] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[72] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[73] NXP (Freescale Semiconductor). *MC1322x Advanced ZigBee™- Compliant Platform-in-Package (PiP) for the 2.4 GHz IEEE® 802.15.4 Standard*, document number: mc1322x edition. Rev. 1.3 10/2010, https://www.nxp.com/docs/en/data-sheet/MC1322x.pdf.

[74] J. Obermaier and S. Tatschner. Shedding too much light on a microcontroller's firmware protection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17). USENIX Association*, 2017.

[75] D. Oktavianto and I. Muhardianto. *Cuckoo malware analysis*. Packt Publishing Ltd, 2013.

[76] I. Pustogarov, T. Ristenpart, and V. Shmatikov. Using program analysis to synthesize sensor spoofing attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 757–770. ACM, 2017.

[77] A. Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided De-*

sign, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016, pages 161–168, 2016.

[78] A. Reid. ARM releases machine readable architecture specification. Blog Post, 2017. https://alastairreid.github.io/ARM-v8a-xml-release/.

[79] A. Reid. Who Guards the Guards? Formal Validation of the Arm V8-m Architecture Specification. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2017.

[80] G. T. H. G. Roussel-Tarbouriech, N. Menard, T. True, et al. Methodically defeating nintendo switch security. *arXiv preprint arXiv:1905.07643*, 2019.

[81] A. Segger. J-link. *Api,. Users guide of the J-Link application program interface (API), Version*, 3:18.

[82] C. Semiconductor. Cyusb301x, cyusb201x ez-usb fx3 superspeed usb controller datasheet [r/ol]. *Cypress Semiconductor*, 2016.

[83] K. Serebryany. Sanitize, Fuzz, and Harden Your C ++ Code. *USENIX Security*, 2015.

[84] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov. Memory tagging and how it improves c/c++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018.

[85] A. L. Shaw, B. Bordbar, J. Saxon, K. Harrison, and C. I. Dalton. Forensic virtual machines: dynamic defence in the cloud via introspection. In *2014 IEEE International Conference on Cloud Engineering*, pages 303–310. IEEE, 2014.

[86] B.-Y. Shen, J.-Y. Chen, W.-C. Hsu, and W. Yang. Llbt: an llvm-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 51–60. ACM, 2012.

[87] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[88] W. Snyder. Verilator: the fast free verilog simulator. *URL: http://www.veripool.org*, 2012.

[89] J. Strömbergson. Hardware implementation of the sha-256 cryptographic hash functions. *Github*.

[90] M. Süßkraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, and C. Fetzer. Prospect: A compiler framework for speculative parallelization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 131–140, New York, NY, USA, 2010. ACM.

[91] W. R. Systems. https://www.windriver.com/products/vxworks/.

[92] T. S. Taft. *Ada 95 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652: 1995 (E)*, volume 8652. Springer Science & Business Media, 1997.

[93] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 291–307, 2018.

[94] C. L. Wang, B. Yao, Y. Yang, and Z. Zhu. A survey of embedded operating system. *Techical Report, University of California, San Diego, USA*, 2001.

[95] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.

[96] Xilinx. Chipscope pro. https://www.xilinx.com/products/design-tools/chipscopepro.html.

[97] Xilinx. Virtual input/output (vio). https://www.xilinx.com/products/intellectual-property/vio.html.

[98] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. *Proceedings of the 2014 Network and Distributed System Security Symposium*, 2014.

[99] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas. Implementation and implications of a stealth hard-drive backdoor. In *Proceedings of the 29th annual computer security applications conference*, pages 279–288, 2013.

[100] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 6:37302–37313, 2018.

[101] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1099–1114, 2019.

SORBONNE UNIVERSITÉ

EURECOM
Sophia Antipolis

RESUME DE THESE DE DOCTORAT DE SORBONNE UNIVERSITE

Spécialité «Informatique, Télécommunication et Électronique»

École Doctorale EDITE de Paris (ED130)

Présentée par Nassim Corteggiani
Directeur de thèse: Prof. Aurélien Francillon

Sujet de la thèse:

Vers une analyse de la sécurité des systèmes embarqués à l'échelle du système.

Introduction.

Les ordinateurs sont de plus en plus présents dans nos vies modernes. Quelques exemples notoires sont les ordinateurs personnels, les téléphones dit intelligents (ou ordiphone), les appareils électroménagers. Ces systèmes vont de l'ordinateur à usage général aux systèmes à usage spécifiques aussi appelé systèmes embarqués.

Historiquement, les systèmes embarqués étaient extrêmement liés aux produits dans lesquels ils étaient intégrés. Ces systèmes étaient conçu pour des usages très spécifiques comme par exemple contrôler la vitesse du moteur d'une machine à laver. En effet, leur usage permet généralement de contrôler un système mécanique (e.g., disque dur) ou électronique (e.g., lecteur optique ou capteur d'image).

Cependant, la récente chute du prix des semi-conducteurs a permis une amélioration notable de ces systèmes avec une montée en complexité et connectivité. Un exemple majeur de cette évolution est l'internet des objets où les objets du quotidien sont connectés à des services en ligne via le réseau Internet. Cette connectivité des objets est de plus en plus importante en connectant par exemple les voitures, les villes, les montres, les appareils électroménagers, les dispositifs de santé. D'après Gartner, en août 2019 4.81 milliards d'objets connectés auraient été déjà fabriqués.

Cette forte demande a permis une forte expansion du marché des semi-conducteurs mais aussi un renforcement de la concurrence avec la présence toujours plus forte de nouveaux acteurs. Cette tension n'est pas sans conséquence sur les cycles de développement qui sont de plus en plus courts pour répondre à la concurrence. De plus, la complexité montante de ces systèmes les rendent de plus en plus difficiles à vérifier afin de prévenir d'éventuelles failles de sécurité. Il est primordial d'effectuer cette vérification avant la mise en production car certains éléments des systèmes embarqués ne peuvent être mis à jour en post-production. C'est le cas des mémoires mortes et des composants matériels.

En effet, certains éléments des systèmes embarqués tel que le code en mémoire morte ou les circuits électroniques ne peuvent pas être mis à jour facilement. Par exemple, les mémoires mortes qui sont programmées lors de la production de la puce électronique par photolithographie. Ce procédé permet d'obtenir des mémoires à faible un coût en comparaison avec d'autres solutions, cependant ces mémoires sont inaltérables. Généralement, le chargeur d'amorçage est stocké sur ce type de mémoire afin d'éviter toute altération de son contenu, ce qui permet également de fournir une racine de confiance. Cependant, si le logiciel n'est pas suffisamment

testé une faille de sécurité logicielle devient alors persistante est difficilement corrigeable sans fabriquer une nouvelle puce. Il en est de même pour les composants matériels. C'est pourquoi les bugs sur ce type de composants sont généralement critiques et peuvent engendrer des conséquences dramatiques.

Les conséquences d'une vulnérabilités peuvent être dramatiques. En effet, les systèmes embarqués contrôlent souvent des structures physiques dans le monde réel dont le dysfonctionnement peut engendrer des dommages humains, matériels ou financiers. Quelques exemples de ces systèmes sont les systèmes de transport de passagers, les robots industriels, les feux de circulation. De plus, ces systèmes peuvent collecter des informations personnelles issues de capteur ou d'interactions avec d'autres objets connectés et de ce fait peuvent révéler des informations personnelles tel que les habitudes, les emplacements géographiques, les intérêts ou même des numéros de carte bancaire. Cette interconnection entre plusieurs appareils identiques peut aussi amener à la construction d'attaques à grande échelle.

Pour toutes ces raisons, il est nécessaire de tester scrupuleusement les systèmes embarqués avant leur mise en production afin de prévenir la présence de faille de sécurité.

Contexte.

Dans cette thèse, nous prenons le point de vue d'un fabricant de puces qui s'intéresse au test des puces avant leur fabrication. Dans un soucis de clarté, nous fournissons une description de ce processus. Généralement, le coeur est conçu par un tiers (par exemple, ARM). Ensuite, ce coeur est étendue avec des blocs matériels personnalisés et réutilisables également appelés bloc de propriété intellectuelle. Ces blocs forment les périphériques matériels qui offrent une porte vers le monde réel. Le niveau de personnalisation de ces blocs peut être important, en particulier pour les circuits intégrés à application spécifique (ASIC). Dans ce cas, les blocs sont personnalisés pour répondre à des besoins spécifiques. Pendant le cycle de développement des puces, le matériel et le logiciel sont conçus simultanément pour s'adapter aux contraintes de temps. Pour cette raison, les fabricants de puces testent généralement les programmes de micrologiciel et les périphériques matériels séparément sur une plate-forme d'émulation ou de simulation. Cependant, tester les composants séparément est souvent inefficace pour détecter les défauts dus aux interactions de différents composants tels que le matériel et le micrologiciel. Pour cette raison, effectuer des tests à l'échelle du système est crucial mais difficile en raison du court délai de mise sur le marché et du manque d'outils dédiés. Pour toutes ces raisons, il existe un besoin d'outils de test de sécurité pour tester minutieusement les logiciels de systèmes embarqués en pré-production.

Problématique.

La complexité croissante des systèmes embarqués a été possible grâce à la combinaison de périphériques matériels et de micrologiciel au sein d'une puce électronique. Ces périphériques offrent soit une interface avec le monde réel, un calcul accéléré ou des fonctionnalités personnalisées. Par analogie, les périphériques matériels sont ce que sont les bibliothèques de logiciels pour les applications de bureau. Ils peuvent avoir une sémantique complexe et des interactions complexes avec le micrologiciel. De plus, ces interactions peuvent être à l'origine de bogues critiques pouvant avoir des impacts dramatiques dans le monde réel. En informatique, des travaux antérieurs ont mis en évidence des perspectives prometteuses pour tester des systèmes complexes à l'aide de techniques d'analyse dynamique ([1], [2], [3], [4], [5], [6], [7], [8], [9]). La recherche dans ce domaine est entravée par le manque de méthodes pour appliquer ces approches sur les systèmes embarqués et encore plus pour l'analyse basée sur les sources. De plus, la plupart de ces solutions traitent uniquement des tests binaires. En particulier, ils s'appuient souvent sur des tests en source fermée uniquement et limitent l'analyse à certains composants des systèmes. Dans cette thèse, nous essayons de répondre à cette question: combien d'efforts faut-il pour appliquer une analyse dynamique à l'échelle du système aux systèmes embarqués tout en considérant un contexte industriel?

Plan de la thèse.

Dans le chapitre 4 «Etat de l'art», un aperçu de l'état de l'art dans le domaine de l'analyse de la sécurité basée sur la source des systèmes embarqués est présenté; dans le chapitre 5 «Évaluation de la sécurité SoC: réflexions sur la méthodologie et les outils», nous présentons une réflexion sur l'analyse de sécurité des systèmes embarqués, et nous soulignons les principaux défis que nous cherchons à relever dans cette thèse; au chapitre 6 «Stéroïdes», une sonde haute performance pour optimiser les approches existantes d'analyse dynamique dédiées à la sécurité; les travaux présentés au chapitre 7 "Inception: Tests de sécurité à l'échelle du système des logiciels de systèmes embarqués du monde réel" illustrent une nouvelle approche pour prendre en charge l'analyse de microprogrammes du monde réel où en pratique différents niveaux sémantiques sont mélangés (par exemple, code assembleur, C / C ++ et code binaire); dans le chapitre 7 "HardSnap: tirer parti du matériel pour les tests de sécurité des systèmes embarqués", nous présentons une technique pour prendre des instantanés à la fois du micrologiciel et des périphériques matériels pour permettre des techniques d'analyse dynamique avancées sans inconsistances et avec des performances raisonnables. La thèse se termine au chapitre 9 avec la conclusion et les perspectives d'avenir.

Évaluation de la sécurité des systèmes sur puce: réflexions sur la méthodologie et les outils.

Dans le chapitre 3, nous présentons une réflexion sur l'évaluation de la sécurité des systèmes sur puce (SoC). En particulier, nous prenons comme référence le concours de sécurité Hack@DAC de 2019 auquel nous avons participé. Ce concours propose à ces participants d'analyser la sécurité d'une puce électronique dans des conditions similaire à celles rencontrées dans l'industrie. A savoir, le code-source des composants matériels et logiciels est disponible. Cependant, les épreuves (la phase de qualification et la finale) se déroulent sur une durée très courte similaire aux contraintes de temps dans l'industrie.

Dans ce contexte, un défi majeur consiste à assurer le bon fonctionnement d'un SoC et de son logiciel, tout en satisfaisant des exigences strictes en termes de fonctionnalités, de coût et de délai de commercialisation. Dans ce chapitre, nous présentons la méthode qui nous a permis de remporter la première place (du classement académique), mais aussi les principaux défis dans ce domaine de recherche.

Générer des tests.

Une tâche qui nécessite un effort manuel est la génération de tests. Généralement, les analystes doivent lire la spécification de sécurité (lorsqu'elle existe), puis souvent l'interpréter pour obtenir une formulation plus précise des propriétés de sécurité souhaitées. Ce n'est qu'alors qu'ils peuvent formuler des hypothèses dans les cas où le système ne répond pas aux exigences, et enfin concevoir le logiciel de test correspondant. En général, il est plus facile d'obtenir des tests significatifs à partir d'une spécification, car celle-ci décrit plus précisément les propriétés attendues du système. À l'extrême, une spécification lisible par une machine pourrait être automatiquement traduite en cas de test. Des descriptions moins précises laissent une marge d'interprétation et nécessitent donc l'utilisation de connaissances spécialisées pour les tests de sécurité.

Une fois qu'une formulation des propriétés de sécurité est disponible, elle peut être utilisée pour évaluer si la mise en œuvre répond aux exigences. Les programmes de test conçus manuellement, tels qu'ils sont utilisés dans notre méthodologie, ne sont qu'une option. Dans ce contexte, il convient de mentionner l'exécution symbolique et le fuzzing, qui sont deux approches populaires dans le domaine des tests de logiciels: ils vérifient les logiciels en explorant automatiquement de nombreux chemins possibles à travers un programme. Cependant, s'il est relativement facile

d'exprimer des exigences de sécurité dans un scénario uniquement logiciel, il n'en va pas de même lorsque des composants matériels éventuellement défectueux entrent en jeu. Nous pensons qu'une bonne spécification pourrait être utilisée pour guider l'analyse effectuée par de tels outils, ce qui permettrait de surmonter la difficulté de définir le comportement attendu. Par exemple, l'exécution symbolique pourrait explorer diverses interactions avec le matériel, tout en vérifiant que les affirmations de sécurité mises en avant par la spécification restent vraies dans chaque cas testé. C'est ce que nous proposons dans les chapitres 5 et 6 de la thèse.

Exécuter les tests de manière efficace.

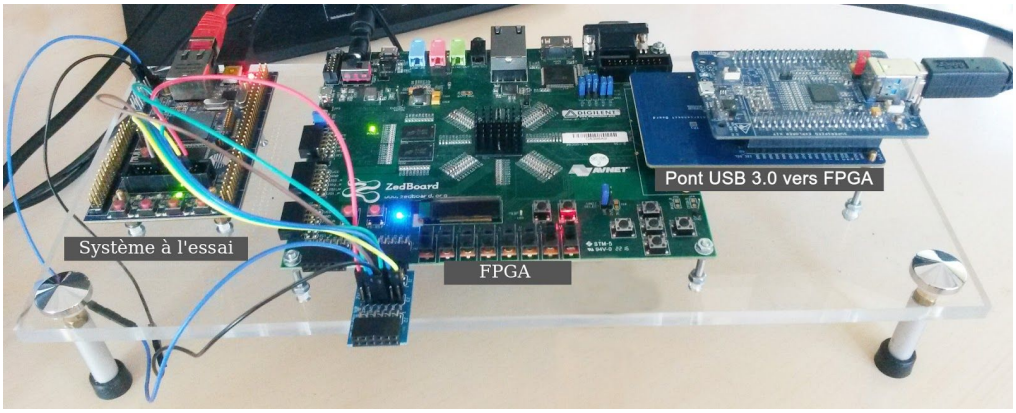
Nous pensons que les tests et la validation d'un SoC peuvent grandement bénéficier de deux approches - abstraction des composants matériels et transfert sélectif vers le vrai matériel tout en utilisant un émulateur. En outre, les capacités des outils existants pourraient être facilement étendues pour coopérer avec des périphériques ou d'autres blocs au niveau RTL. Nous présentons cette approche au chapitre 8. Ces outils étendus serviraient alors de plate-forme pour les tests de sécurité des SoC basés sur des logiciels. Ils ont déjà été conçus en mettant l'accent sur la sécurité et comprennent un certain nombre de contrôles de sécurité automatisés (par exemple, des contrôles de corruption de la mémoire). En outre, ils permettent des techniques d'exploration plus automatisées telles que l'exécution symbolique et le "fuzzing".

Stéroïdes: un débogueur USB à faible latence rapide.

Dans le Chapitre 4, nous présentons un système de communication appelé Steroids qui permet la réalisation d'une plate-forme de test découplant le logiciel et le matériel tout en conservant un interaction rapide entre les deux. Steroids se base sur le protocole USB 3.0 afin de garantir un latence acceptable. Son design est basé sur des composants sur étagère, abordables et son code-source est ouvert. Ainsi, les résultats de nos expériences sont reproductibles et Steroids peut servir de base à d'autres projets de recherche.

Ce système est notamment utilisé dans le chapitre 5, où nous présentons une plate-forme de test pour micrologiciel. Et dans le chapitre 6, où nous proposons une amélioration de cette plate-forme avec notamment un meilleur prise en charge des composants matériel.

Pour finir, nous présentons les améliorations en terme de temps d'analyse sur un outil d'analyse de code binaire nommé avatar². L'utilisation de Steroids sur cet outil permet de diviser par 16 la latence lors des accès en écriture et lecture vers le matériel.



Photographie du système Stéroïdes avec de gauche à droite: le système embarqué en cours de test, le FPGA (ZedBoard) avec un JTAG maître et le pont USB 3.0.

Inception: test de sécurité à l'échelle du système des logiciels de systèmes embarqués du monde réel.

Dans ce chapitre, nous présentons Inception, un outil permettant d'effectuer des tests de sécurité sur des microprogrammes complets. Inception introduit de nouvelles techniques d'exécution symbolique pour les systèmes embarqués. En particulier, un compilateur génère et fusionne du code LLVM à partir de code binaire et de code source de haut niveau. La présence de différents niveaux de sémantique dans un microprogramme est fréquent. En effet, des lignes d'assembleur sont souvent mélangées avec du code plus haut niveau (par exemple, le langage C/C++) afin d'interagir avec le matériel sous-jacent. De plus, des librairies logicielles peuvent être fournies au format binaire uniquement. Notre système permet la prise en charge de ce type de code où différents niveaux de sémantique co-existent. La génération d'un modèle sémantique en LLVM IR permet l'interopérabilité avec l'ensemble des outils d'analyse supportant ce format qui est devenu commun aujourd'hui.

Inception dispose aussi d'une machine d'exécution symbolique, basée sur Klee, qui effectue une exécution symbolique du LLVM IR, en utilisant plusieurs stratégies pour gérer différents niveaux d'abstraction mémoire et l'interaction avec les périphériques (par exemple, les interruptions et la mémoire mappée). Enfin, nous intégrons Steroids, un débogueur JTAG haute performance qui redirige les accès mémoire vers le matériel réel.

Nous avons validé notre implémentation à l'aide de 53 000 tests comparant l'exécution d'Inception à l'exécution concrète sur une puce Arm Cortex-M3. Nous montrons ensuite les avantages de Inception sur un benchmark composé de 1624 programmes vulnérables synthétiques, de 4 applications industrielles et open source, et de 19 démonstrations. Nous avons découvert 8 crashs et 2 vulnérabilités inconnues jusqu'alors, démontrant ainsi l'efficacité d'Inception en tant qu'outil d'aide aux tests de micrologiciels de dispositifs embarqués.

HardSnap: Un framework pour le test de système embarqué basé sur les instantanés du matériel.

Les techniques d'analyse dynamique avancées telles que le fuzzing et l'exécution symbolique dynamique (DSE) sont une pierre angulaire des tests de sécurité des logiciels et sont de plus en plus utilisées pour les tests de systèmes embarqués. Le test de logiciels dans une machine virtuelle offre une meilleure visibilité et un meilleur contrôle. Les instantanés de la machine virtuelle permettent également de gagner du temps lors des tests en facilitant la reproduction des pannes, en effectuant une analyse des causes profondes et en évitant de ré-exécuter les programmes dès le départ. Toutefois, les systèmes embarqués étant des machines très diverses, il est souvent impossible de les émuler parfaitement. Les travaux antérieurs dans ce domaine tentent soit de modéliser le matériel, soit d'effectuer une émulation partielle (voir chapitre 7), ce qui entraîne une émulation souvent imprécise ou lente.

Dans ce chapitre, nous proposons donc une nouvelle approche, appelée HardSnap, pour tester conjointement le matériel et les logiciels avec un niveau élevé d'introspection. HardSnap vise à améliorer les tests de sécurité des systèmes conçus conjointement par le matériel et les logiciels, où les concepteurs de systèmes embarqués ont accès à l'ensemble de la pile HW/SW. HardSnap est une solution basée sur une machine virtuelle qui étend la visibilité et la contrôlabilité des périphériques matériels avec un surcoût négligeable. HardSnap introduit le concept d'un instantané du matériel qui recueille l'état du matériel (ainsi que l'état du logiciel). Dans notre prototype, les blocs matériels en Verilog sont soit simulés dans un logiciel, soit synthétisés sur un FPGA. Dans les deux cas, HardSnap est capable de générer un instantané du HW/SW à la demande. HardSnap est conçu pour prendre en charge de nouveaux périphériques de manière automatique, pour avoir des performances élevées et une contrôlabilité et une visibilité totales sur les logiciels et le matériel. Nous avons évalué HardSnap sur des périphériques open-source et des micrologiciels synthétiques pour démontrer une meilleure capacité à trouver et à diagnostiquer les problèmes de sécurité.

Conclusion.

Cette thèse présente des améliorations fondamentales pour réaliser une analyse de la sécurité des logiciels embarqués à l'échelle du système mais aussi des composants matériel. Nous explorons le problème de l'analyse de la sécurité des systèmes embarqués tout en considérant le point de vue d'un analyste de sécurité ayant accès au code source et aux langages de description du matériel (HDL). Nos travaux ont permis la réalisation d'une plateforme de test qui permet de surmonter certains des problèmes majeurs qui entravent la recherche dans ce domaine. A savoir, l'analyse de code logiciel ayant différent niveaux de sémantiques, la mise en place d'une communication à faible latence pour les outils utilisant l'émulation partielle et enfin l'exécution concurrente de différent chemin d'exécution d'un microprogramme tout en conservant un cohérence avec l'état du matériel.

L'ensemble des travaux présentés dans cette thèse sont disponible sous des licences libres afin de garantir la reproductibilité des expériences mais aussi de promouvoir la recherche ouverte.

Références.

[1] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. FIE on firmware:Finding vulnerabilities in embedded systems using symbolic execution. In USENIX Security Symposium, pages 463-478, 2013.

[2] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, andZ. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In 49th USENIX Security Symposium, pages 291-307, 2018.

[3] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.Proceedings of the 2014 Network and Distributed System Security Symposium, 2014.

[4] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. Butler.Firmusb: Vetting usb device firmware using domain informed symbolic execution. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2245–2262. ACM,2017.

[5] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In 28th USENIX Security Symposium, pages 1099-1114, 2019.

[6] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna.Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In NDSS, 2015.

[7] K. Koscher, T. Kohno, and D. Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In WOOT, 2015.

[8] D. D. Chen, M. Woo, D. Brumley, and M. Egele. Towards automated dynamic analysis for linux-based embedded firmware. pages 1–16, 2016.

[9] M. Süßkraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, and C. Fetzer. Prospect: A compiler framework for speculative parallelization. In Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '10, pages 131-140, New York, NY, USA, 2010. ACM.