HardSnap: Leveraging Hardware Snapshotting for Embedded Systems Security Testing

Nassim Corteggiani, Aurélien Francillon
EURECOM
nassim.corteggiani@eurecom.fr, aurelien.francillon@eurecom.fr

***Abstract*—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 (forwarding interaction to the real hardware), 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 paper, 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.

***Index Terms*—Embedded Systems, Hardware Snapshotting, Security Analysis, Symbolic Execution**

I. 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 microcontrollers 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, embedded 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 [16], [30], [31], symbolic execution [5], [25], forensics analysis [13], [17], [21], and malware analysis [20], [26]. 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 face 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 [4] or automated [3], [8], [12] 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 [29], [15], [9], [24]. 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. The tester may have access to the peripherals source code which provides 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 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. [24] 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 [22], [6], [11] 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 may slowdown the dynamic analysis.

In this paper, 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 [9], 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 hardware 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 subsystem. 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 paper 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.

A. Related Work

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

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 [7] that is based on QEMU [4]. S2E enables symbolic execution while emulating peripherals through behavioral models written in 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 [29]

	Over-Approximation	Sub-Approximation			Full Emulation			Partial Emulation			Simulation	Hybrid	
	FIE [10]	P²IM [3]	HALucinator [8]	Pretender [12]	Integrated Logic Analyzer (FPGA) [28] [27]	FPGA	S2E [7]	Avatar1/2 [29] [18]	Inception [9]	Surrogates [15]	Verilator [22]	QEmu+System-C [6]	HardSnap
Abstraction Level	B[a]	B[a]	B[a]	B[a]	P	P	B	B/P	P	P	L	B/L	B/L/P
Symbolic Execution	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✓
Full Visibility	✗[b]	✗[b]	✗[b]	✗[b]	✗(Limited scope)	✗	✓	✗	✗	✗	✓	✓	✓
Full Controllability	✗[b]	✗[b]	✗[b]	✗[b]	✗(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 I: Comparison of HardSnap with the related work. L: Logical (RTL level) P: Physical B: Behavioral
[a] No Hardware Interactions.
[b] Using either a sub-approximation or an over-apprimatation.

redirects hardware interactions to the real device. This method has later been followed by Surrogates [15] and Inception [9]. The former and the later 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 [10] or a sub-approximation [3], [8], [12]. 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 [22], [2] 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. [6] 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.

II. 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 complete. 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. [19], 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. 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 [19]. 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 [9], [29]. These tools evaluate concurrently different execution path 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.

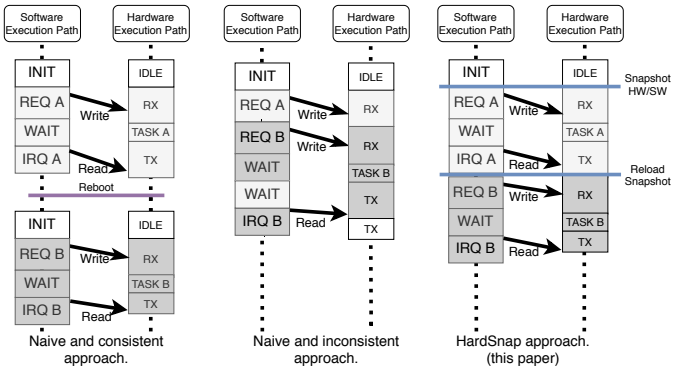

Fig. 1: Description of different Hardware/Software co-testing execution. From left to right: naive and consistent but slow approach; naive and fast but inconsistent approach, HardSnap approach.

III. 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, which 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 [9], a framework for firmware program analysis based on the Klee [5] symbolic execution engine. We emphasize that our approach is not specific to Inception, and can be extended to any hardware-in-the-loop dynamic

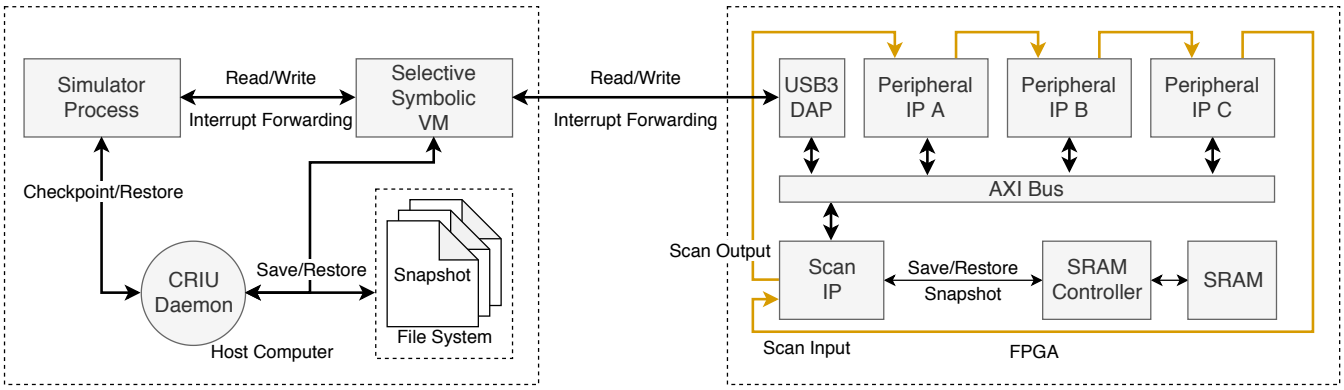

Fig. 2: Overview of HardSnap.

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 of the peripheral state. To guide the reader during our explanation, we provide a description of HardSnap in Fig. 2.

A. 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 synchronize the circuit operations at each 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 a good candidate 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 automatically 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) Tar-

get. We designed a second hardware target that focuses on performance at the cost of full execution tracing. We present this target on the right side of Fig. 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 very 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 lack advanced debugging 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 V.

B. Selective Symbolic Virtual Machine

In this context, HardSnap has been implemented on top of Inception [9], 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 [7]. 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 a JTAG debugger. We extended this mechanism to our simulator and FPGA target. We developed custom driver 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).

C. Snapshotting Controller

In Fig.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 it 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.

IV. 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.

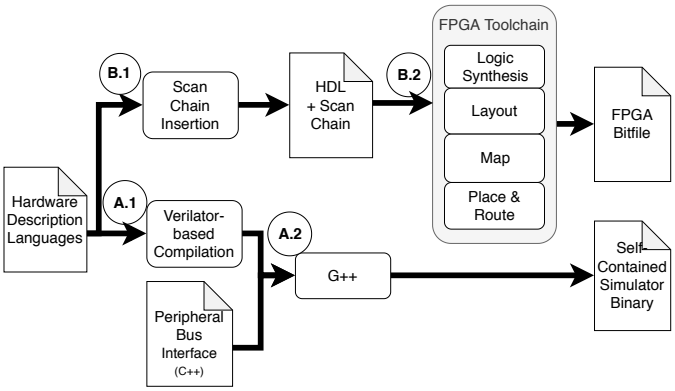

Fig. 3: HardSnap's instrumentation toolchain.

A. Hardware Snapshotting Instrumentation

To support small and complex hardware designs alike, we use two approaches (Fig. 3): purely software simulation or FPGA backed simulation.

Simpler hardware components can be simulated purely by software. For this, we extend `Verilator` [22], an open-source Verilog simulator designed 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).

B. 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}. Figure 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 $S_{previous}$ keep 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,R,S_previous);
5      if S_previous ≠ ∅ and S ≠ S_previous then
6          UpdateState(S_previous );
7          RestoreState(S);
8          S_previous = S;
9      end
10     S_previous = S;
11     ServePendingInterrupt(S);
12     StepInstruction(S);
13     S_new = ExecuteInstruction(S);
14     AS ⇐ AS ∪ S_new;
15 end
```

V. 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.

Case study: testing the PIC peripheral. 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* [23] 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 [1] that is deployed on commercial HSM.
- *AES Counter Mode* [14]. 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.

A. Experiment I: Hardware Snapshotting Performance

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

a) 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 4. Additionally to this experiment, we provide in table Table II 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 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 II: Size of the test design corpus.

b) 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 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.

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

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 experiment 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.

B. 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.

a) 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 level 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

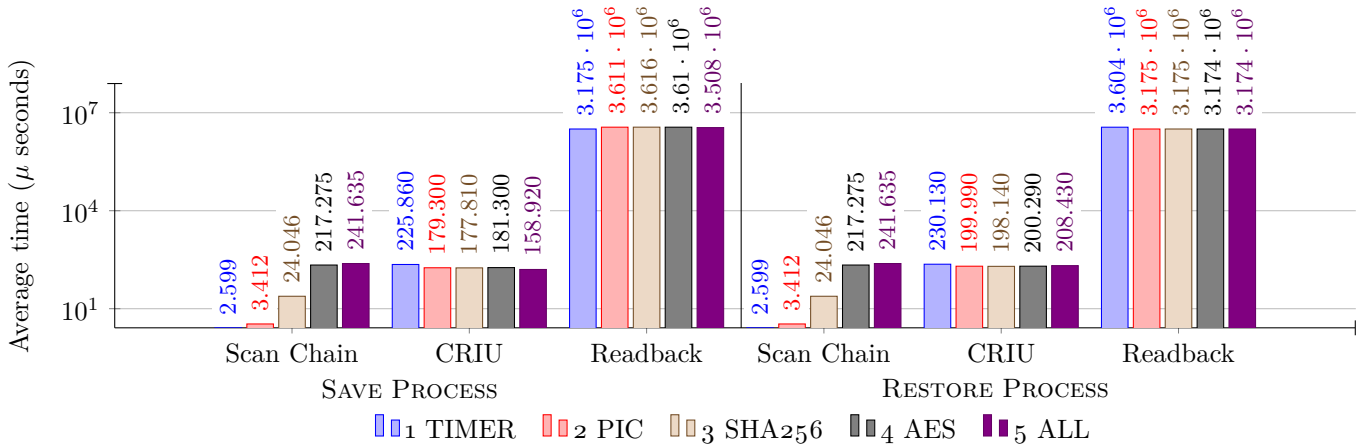

Fig. 4: Average duration, in microseconds, for 10^6 snapshot saves or restores for the FPGA and the simulator. Note the y-axis is plotted on a logarithmic scale.

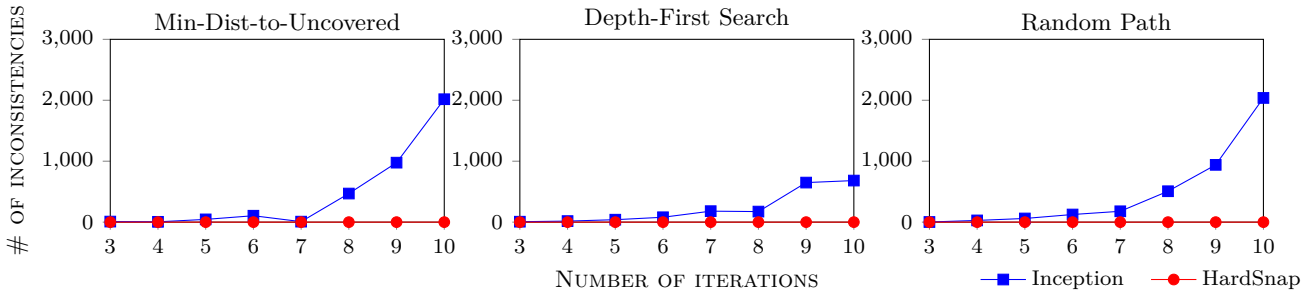

Fig. 5: The cumulative number of inconsistencies found in a synthetic firmware with different state selection heuristics by the number of iterations.

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 5.

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. [24]. 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 [9]. Instead of restarting the execution from zero (which is extremely slow), we choose to implement the record-and-replay mechanism from Avatar [29]. We report the results in Figure 6.

b) 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 correctness 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. Futhermore, 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.

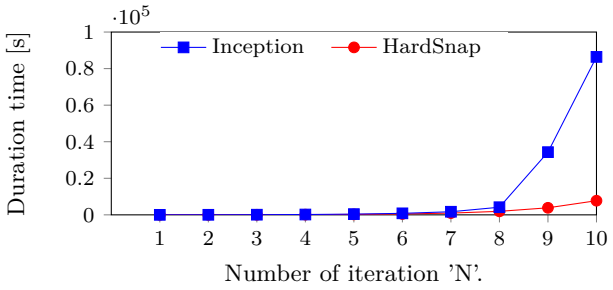

Fig. 6: The duration time by the number of iterations 'N' (Number of explored states=2^N). Note the y-axis is plotted on a logarithmic scale.

C. 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 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, 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");
}
```

Fig. 7: Use case written in C to verify the correctness of the PIC peripheral.

VI. 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.

VII. Conclusion

In this paper, 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 bottle neck 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/.

References

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

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

[3] 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.

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

[5] Cadar, C., Dunbar, D., Engler, D. R., et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[6] Chiang, M., Yeh, T., and Tseng, G. 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 (April 2011), 593–606.

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

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

[9] Corteggiani, N., Camurati, G., and Francillon, A. Inception: System-wide security testing of real-world embedded systems software. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association, pp. 309–326.

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

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

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

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

[14] Hsing, H. Advanced encryption standar FPGA implementation.

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

[16] Maier, D., Radtke, B., and Harren, B. 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.

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

[18] Muench, M., Nisi, D., Francillon, A., and Balzarotti, D. Avatar2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)* (2018), vol. 18, pp. 1–11.

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

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

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

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

[23] StrÃűmbergson, J. Hardware implementation of the SHA-256 cryptographic hash functions. *Github.*

[24] Talebi, S. M. S., Tavakoli, H., Zhang, H., Zhang, Z., Sani, A. A., and Qian, Z. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, Aug. 2018), USENIX Association, pp. 291–307.

[25] Wang, F., and Shoshitaishvili, Y. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)* (2017), IEEE, pp. 8–9.

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

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

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

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

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

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