

DSCOPE: To Reliably and Securely Acquire Live Data from Kernel-Compromised ARM Devices

Zhe Chen, Haiqing Qiu, and Xuhua Ding

Singapore Management University
{zhechen, hqqiu, xhding}@smu.edu.sg

Abstract. Live data acquisition from a mobile device controlled by a corrupted kernel is challenging as the adversary can block data reporting from the inside and also sabotage external I/O interactions. This paper proposes DSCOPE as a reliable live data acquisition system for ARM devices without trusting their kernels. It ensures that a device user can always launch DSCOPE to securely extract the needed virtual memory data when the device is under attack. Besides its reliability, DSCOPE also preserves kernel semantic and support user-customized acquisition routines. We have built a prototype of DSCOPE on a Raspberry Pi 4 development board and have also tested DSCOPE’s reliability against various forms of denial of service attacks. Our experiments show that a user can dynamically import data acquisition routines to the device to extract kernel objects and runtime stacks from an attack scene or a kernel crashing site.

1 Introduction

Live data acquisition is integral to incident response, forensics and even crash diagnosis as it provides the raw material for software semantics derivation and attack intelligence extraction. Since data acquisition is often in the presence of an adversary, the procedure is required to be *secure*. Namely, the data is retrieved genuinely from locations demanded by the user. The procedure should not be spoofed or tampered with by the adversary in the device. Equally important to security is the assurance of data retrieval with the same logical-to-physical location mappings as used by the kernel. We term it as *mapping consistency*. For volatile data, the logical location refers to the virtual address in the kernel code referencing the object. An acquisition tool using mappings different from the target may fetch data from wrong physical pages and data analysis cannot be made through the same perspective as the kernel.

The challenges to attain security and mapping consistency in data acquisition mirror those in virtual machine introspection (VMI). In the literature, VMI can be classified into in-VM introspection [22, 24] and out-of-VM introspection [8, 10, 20]. As noted in [4, 15], the former technique uses a consistent mapping, however, without security assurance against a compromised kernel. While the latter can cope with the kernel adversary, it has no guarantee of mapping consistency. Similarly, a device’s runtime data can be acquired in-device by relying on a

kernel module or out-of-device by using a special external hardware equipment attached, e.g., a JTAG debugger. The conflicting requirement between security and mapping-consistency in VMI also arises in device data acquisition when the adversary resides in the kernel. The adversary can tamper with the in-device agent to fake the reported data or uses its own address mappings to explore the semantic gap [15], thus undermining the soundness of the collected data. TrustDump [25] overcomes the challenge by freezing the untrusted kernel and walks through its paging hierarchy using an agent protected by ARM TrustZone. While suitable for page-granularity memory dump, it is not efficient and feasible for fine-grained live memory object acquisition, e.g., observing the dynamic of cred objects of all processes during corrupted kernel execution.

A remarkable difference between device and VM data extraction lies in *reliability*. Namely, a data acquisition procedure can always be launched and carried out against denial-of-service attacks. High reliability is a desirable feature for live forensics and incident response. Several schemes [13, 30, 31] rely on a hardware watchdog or timer to ensure a security-critical task to be reliably launched. Nonetheless, we consider on user demand data acquisition, which makes periodical checking less appealing. More importantly, these schemes do not need to consider runtime denial-of-service attacks since the launched task controls the entire platform. GAROTA [2] assures a reliable launch by protecting the interrupt delivery route in micro-control units which have a simpler architecture than middle or high-end ARM devices such as phones.

In this paper, we propose DSCOPE, a virtualization based system for live data acquisition. DSCOPE uses a daemon running in EL2, with the size orders of magnitude smaller than a full-fledged hypervisor like Xen-on-ARM [14]. The device user can reliably launch a data acquisition session according to her demand in the presence of a kernel adversary. Without terminating the kernel or shutting down the device, the user securely connects the device with her trusted PC via a serial cable and imports her data acquisition agent to the device. The agent’s execution uses the kernel’s current address mappings to read/write kernel memory and transports the results to the PC. DSCOPE ensures that the agent import and execution are not subject to the kernel’s attacks, including denial-of-service attacks. We have implemented a prototype of DSCOPE on a Raspberry Pi development board with an ARM64 processor. We assess its reliability by running several experiments including benign kernel crashes as well as denial-of-service attacks that attempt to sabotage I/O interfaces, block interrupt delivery, and withhold CPU cores. All experiments show that DSCOPE is reliable and practical, with an insignificant performance overhead upon a normal device.

2 Background and Related Work

2.1 Background on ARM Virtualization

ARM virtualization extension supports physical memory virtualization, interrupt virtualization and timer virtualization. ARMv7 and subsequent versions use the Stage-II address translation tables for the hypervisor to define mappings

between intermediate physical addresses (IPA) to physical addresses (PA), while the kernel uses the Stage-I address translation tables to map virtual addresses (VA) to IPA. The hypervisor can control the kernel’s access to memory by setting stage-II page table. Once memory virtualization is enabled, a memory access requires traversal of both translation tables.

The Generic Interrupt Controller (GIC) [3] is a centralized resource for supporting and managing interrupts in a system that includes at least one processor. As a central controller, GIC can disable, enable or generate processor interrupts from hardware interrupt source. It is compatible with multiprocessor environments and ARM architecture virtualization extensions. Logically, the GIC system includes these two functional components: Distributor and CPU interfaces. The Distributor centralizes all interrupt sources, determines the priority of each interrupt, and for each CPU interface, forwards the interrupt with the highest priority to the associated CPU core. GICv2 and subsequent versions support for ARM interrupt virtualization with which the hypervisor can intercept and dispatch interrupts to the kernel.

2.2 Related Work

Our work is related to software reliability against component failures. The popular approach to reliability is Software Fault Isolation (SFI) [28] with the idea that errors are contained within certain region of the system. For instance, several works [9, 11, 17] use memory isolation as the building block. It is more challenging to consider failures due to attacks in system software. Nooks [26] uses a copy version of kernel page table for untrusted kernel extension which is only endowed read permission on kernel space so that misbehaving kernel extension cannot directly modify kernel data. ARMLock [34] uses domain access control register (DACR) to create multiple sandboxes for isolating each software or kernel extension. AION [1] provides availability for enclave code execution by introducing a new instruction which ensures an atomic execution for a number of CPU cycles. Cider [31] and Lazarus [13] propose authenticated hardware watchdog to ensure a compromised IoT device to be reset. RT-TEE [30] uses a secure timer to ensure that tasks in the ARM Secure World get the time slices. GAROTA [2] protects the secure interrupt delivery to ensure that the root-of-trust code to be launched when the prescribed interrupt is triggered. In the first three schemes, the activate tasks are not subject to attacks due to the hardware protection, whereas GAROTA validates the code integrity of the root-of-trust. Nonetheless, none of them needs to protect the availability of a task running with the same privilege as the adversary.

TrustZone-based Trustdump [25] reliably dumps volatile data in the Normal World. It routes all interrupts to the Secure World which hosts a memory dumping agent. During memory dumping, the kernel in the Normal World is stopped from execution because the device has only one core. To access the kernel’s virtual memory, the dumping agent traverses the paging hierarchy used by the kernel starting from the banked TTBR1 register. Trustdump is more suitable for dumping static memory pages than traversing a kernel object list, due to

the overhead of software-implemented paging hierarchy walking. Moreover, it cannot be used for *live* acquisition where the malicious kernel is still running. SMMdump [29] is applied on x86 platforms only and only considers physical memory dumping instead of virtual memory pages.

In a broader sense, this paper is related to virtualization based secure systems where a full-fledged hypervisor (e.g., KVM [7] and Xen [14]) or a micro-hypervisor dedicated for security (e.g., XMHF [27]) is used to enforce system-wide control over the memory and I/O devices. For instance, SecVisor [21], Lares [19] and Hypernel [16] deals with kernel code integrity. Overshadow [5], InkTag [12] and AppShield [6] consider user-space memory page isolation against an untrusted OS. FIMCE [32] provides a secure execution environment with a pool of isolated memory pages and an isolated CPU core, so that it defeats the stifting attack and the virtual processor ID (VPID) attack. Our work will use techniques in FIMCE [32] as one of the building blocks.

3 The Design of DScope

We begin with the problem formulation and then present the architecture of DSCOPE followed by detailed descriptions of its components.

3.1 Problem Description

We consider the scenario that a user kickstarts a live data acquisition from an IoT device whose kernel either malfunctions or is compromised by malware. For instance, the user activates the session when the device becoming unresponsive to user actions or after receiving an alert raised by an intrusion detection system.

Target Device DSCOPE is designed for mobile and IoT devices using an ARM64 multicore device with virtualization extension (supporting vGIC). The device is equipped with a USB or UART port for external communications. DSCOPE can be exported x86 platforms with modest modification.

Trust Model We trust the device’s hardware and firmware including BIOS, all of which behave as expected. We trust the bootloaders and the associated secure bootup procedure. Namely, the bootloaders ensure the launching time integrity of the hypervisor in the normal world. We also assume that code and data in the Exception Level 2 (EL2) are not susceptible to attacks. We trust the PC used for interrogating the device and receiving the acquired data.

Threat Model The adversary in our consideration is malicious software that runs arbitrary code with kernel privilege on the target device. Its objective is to obstruct, disrupt and tamper with data acquisition. It may launch denial-of-service attacks such as masking interrupts, replacing the original kernel interrupt handler to exclusively occupy CPU cores, and misconfiguring some I/O interfaces to prevent the device from interacting with external PCs. It may attempt to feed faked data to the acquisition tool. Physical attacks are out of scope.

Goals We aim to solve to the problem with *security*, *reliability* and *mapping-consistency*. We also expect the solution scheme to be *performance-friendly*, in the sense that (a) it does not take a significant performance toll to the device’s day-to-day operations; and (b) it can be launched promptly and extracts data at a high speed. This requirement is due to that, despite the recent hardware advances, IoT devices are still more resource constraint than desktop computers.

The main challenge is to achieve four aforementioned goals altogether. A secure data acquisition mechanism can be obtained by using a special external hardware attached to the memory bus or by tasking the hypervisor or software in TEE. Nonetheless, the data extractor does not use the kernel’s address mappings to reference the memory. One might propose a trusted agent in the device to bridge the semantic gap. As noted by Jain et. al. [15] on in-VM introspection schemes [22, 24], the agent is not hardly secure against the kernel adversary, not to mention reliability.

3.2 The Design of DScope

Basic Workflow We design DSCOPE as a hypervisor-based data acquisition scheme realizing all four goals. The basic workflow is as follows. To start an acquisition session, the user generates a hardware interrupt using a button on the device. The interrupt triggers the hypervisor to launch of a fully isolated environment on a CPU core. This environment, termed as Device Operation Environment (DOE), imports a data acquisition agent from the external PC and executes it with the same virtual-to-physical address mappings as the kernel. At any time, the user can respond to attacks by manually triggering the prescribed hardware interrupt. During data acquisition, the hypervisor and the DOE only occupy one CPU core with other cores controlled by the (malicious) kernel.

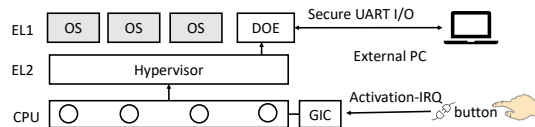


Fig. 1. A system overview of the architecture of DSCOPE.

The Approach By and large, we attain the security and reliability by leveraging the hypervisor’s privilege of configuring the hardware (e.g., GIC) and system settings (e.g., the stage 2 translation tables), whereas the mappings consistency of data acquisition is realized by the system architecture of DOE. Since various virtualization-based techniques for security purposes have been proposed in the literature, our primary challenge is about reliability assurance. Specifically, we face the following challenges: (i) how to ensure that the user can *always* activate the hypervisor to start the acquisition session? (ii) how to ensure that the

acquisition agent’s execution inside the DOE is *not* sabotaged by DoS attacks from the malicious kernel running in other cores?

The solution to these challenges cannot significantly impact the device’s performance in its routine operations. For instance, it is not a satisfactory solution for the hypervisor to persistently hold one CPU core since the device with four cores potentially suffers 25% performance drop. Our basic approach is to leverage several features of ARM vGIC to empower the hypervisor the priority of interrupt filtering. The hypervisor runs as a daemon and stays dormant most of the time. Unlike in x86 platforms, ARM virtualization does not allow selected events to trap the CPU to the hypervisor. Thus, the hypervisor in DSCOPE is “waken” by all interrupts. However, except the activation interrupt, other interrupts are rerouted to the kernel to process to minimize the performance impact.

We cope with the mapping-consistency and reliability of the DOE by improving and combining techniques used in ImEE [33] and FIMCE [32]. The former provides mapping-consistency for an out-of-VM introspection whose system setting does not match DOE. The latter provides a fully-isolated environment using a hypervisor, however, without reliability protection. As show later, we follow the approach in FIMCE to construct the DOE and enhances its reliability by blocking unsolicited interrupts. Within the DOE, we adapt the approach used in ImEE to enclose the kernel’s paging hierarchy for the agent to use.

3.3 The Hypervisor

GIC Configuration During device boot-up, the DSCOPE hypervisor maps the GIC’s CPU interfaces for all cores to the corresponding vCPU interfaces and enables interrupt virtualization. As a result, all external hardware interrupts are delivered to EL2 of core 0. An IRQ number chosen by the user is reserved for data acquisition activation. Note that the choice should allow the user to manually trigger the interrupt at any point of time. We call it the *Activation-IRQ*. A potential denial-of-service attack is to generate a flood of interrupts/exceptions to block the delivery of the Activation-IRQ. To copy with the attack, the hypervisor configures the GIC priority register so that the Activation-IRQ is given the highest priority and all others are lower. As a result, the CPU always responds to the Activation-IRQ without being delayed. To prevent the kernel from tampering with the interrupt setting, the hypervisor sets both the GIC Distributor and the GIC priority register as inaccessible to the kernel by configuring the stage 2 translation tables.

State Transition At runtime, the hypervisor’s execution is triggered by an explicit hypcall from the kernel or an exception/interrupt destined for EL2. Initially, the hypervisor runs in the *inactive* state and enters the *active* state after receiving the Activation-IRQ. It returns to the inactive state after the acquisition session is over. When inactive, it monitors all interrupts delivered to core 0. Except for Activation-IRQ, it handles the interrupt by injecting the corresponding virtual interrupt to the GIC so that the hardware further interrupts the kernel to handle the event. When the Activation-IRQ arrives at core 0, the hypervisor

switches its state and sends an IPI to a randomly chosen secondary core (e.g., core 2) as the DOE core and resumes the kernel. We choose a secondary core for the DOE for two reasons. First, normal hardware events do not disrupt the DOE as they are delivered to core 0. Hence, we only deal with malicious events delivered to the DOE. Secondly, the kernel can still attend to the interrupts delivered to core 0 which is crucial to keep the device alive.

As a result of the IPI sent from the hypervisor, the DOE core is trapped to EL2 and the hypervisor starts its execution there. The DOE consists of an isolated physical core, an isolated memory page frames and a secure I/O interface. To prevent the adversary from preempting the DOE from its core, the hypervisor filters all Software Generated Interrupts (including IPIs) bound for the DOE. The implementations details of how the hypervisor handles the interrupts are presented in Section 4.2.

3.4 Device Operation Environment (DOE)

In DSCOPE, the hypervisor is only for security and dependability assurance. We leave all functionality for data acquisition or other device operation to the DOE so that (1) it avoids bloating the hypervisor code; (2) it allows users to flexibly choose their preferred secure actions according to the ongoing attack.

DOE Components The DOE is a computing environment comprising the D-core, a UART port and a segment of physical memory. The UART port is used by the DOE code to communicate with the user’s trusted PC. The physical memory segment is priorly reserved by the hypervisor.

Architectural Setting The DOE core runs at EL1. Namely, the acquisition agent runs with the kernel privilege and never downgrades itself to the user-privilege. The VA-IPA-PA mappings consists of two parts. The *local* mappings map the DOE’s own code and data to the DOE physical memory segment. VAs in the local mappings are in the lower half of the 48 bits address space and are translated via TTBR0.

The *device* mappings are for the kernel VA range, i.e., the higher half of the 48 bits address space. Instead of copying the kernel’s stage 1 translation tables to the DOE, the hypervisor copies TTBR1 used by the kernel to the DOE core. The hypervisor clones the stage 2 translation table used for the kernel to the DOE core, with all pages’ execution permissions removed. With this setup, the DOE code reads and writes the kernel’s virtual memory in the same view as the kernel without the risk of kernel code being executed in the DOE.

The environment uses a synchronous UART I/O channel with the external PC. We choose UART as the DOE’s I/O port because its control and data interfaces are rather simple. To receive data from the PC, the DOE software continuously polls the UART port. As compared with normal DMA based UART operations, the polling software is much simpler. More importantly, it avoids conflicts between the interrupt handling and the hypervisor’s protection of the DOE.

Code Components Initially, the software loaded to the DOE by the hypervisor consists of the *UART library*, the *manager* and the *default agent*. They are all copied to the physical memory frames reserved during system bootup for the hypervisor and are (re)mapped to the DOE's space upon its creation. The URAR library provides functions for communication with the external PC; the manager provides basic system service such (un)loading an agent and handling exceptions; the default agent offers a primitive acquisition capability, i.e., receiving commands from the PC and returning the needed kernel objects.

Agent Importation & Execution A DOE agent is a chunk of binary code performing the expected device operations. DSCOPE requires the agent to be DOE-contained in the sense that it does not use any code outside of the DOE memory. The agent uses the UART library to communicate with the connected PC and directly references the kernel virtual addresses. An imported agent is loaded to the predefined virtual address. Since the DOE does not enclose a memory manager for dynamic memory (de)allocation, the agent has to manage its own memory layout. the DOE code runs in EL1. The manager can dynamically configure the stage 1 translation tables to provide the needed local mappings.

The agent exclusively occupies the DOE core without being interrupted. It yields the DOE core to the manager after its mission is accomplished. Runtime exceptions, e.g., divided-by-zero, are handled by the manager by default. The agent can register a call-back function for the manager to invoke. Otherwise, the manager unloads the agent and requests the hypervisor to terminate the DOE.

3.5 Protection On DOE

To ensure security and reliability of the DOE, the hypervisor isolates all its hardware resources against the kernel, including the DOE core, the memory pages and the UART port.

Core Isolation The protection on the DOE core prevents the denial-of-service attacks from the kernel which may send IPIs such as an INIT signal to disrupt the agent's execution. In DSCOPE, core isolation means that a physical CPU core is exclusively occupied by DOE when DSCOPE is in active state. Once DSCOPE being activated, the DOE core does not receive external interrupts until it is released by the hypervisor.

Memory isolation Memory isolation blocks any unauthorized access to the physical memory frames used by the DOE. The physical memory frames used by the hypervisor (including the DOE) are reserved during boot up and are not used in stage 2 translation tables for the kernel. Since the kernel cannot modify the stage 2 translation tables, the adversary cannot access those memory frames. Note that before hypervisor launches the DOE, it flushes all TLBs in the DOE core so that the adversary cannot leave any poisonous mappings for the DOE.

I/O isolation I/O isolation ensures that the trusted PC communicates with the DOE with no risk of data leakage, modification or disconnection. It is realized

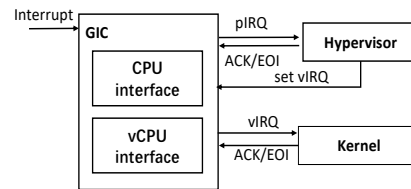
by blocking the kernel’s memory accesses to any UART registers, since all I/O ports in ARM processors use MMIO only. Specifically, when setting up the DOE, the hypervisor unmaps the UART registers from the stage 2 translation tables used by the kernel, and remaps it to the stage 2 tables for the DOE.

4 Implementation

We have implemented a prototype of DSCOPE on a Raspberry Pi 4 development board. The board is equipped with a Broadcom BCM2711B0 processor with 4 cores and 4 GB RAM. Figure 2(a) shows our prototype. Port 1 is the secure UART interface that can be connected to trusted PC using a cable. Port 2 mimics a button in an off-the-shelf device. The two cables are connected to the GND pin and GPIO 04 (which is chosen as the Activation-IRQ) respectively. A user triggers the Activation-IRQ by tapping the ends of the cables together to generate the signal.



(a) A Raspberry Pi 4 board installed with DSCOPE.



(b) DSCOPE’s interrupt handling process.

Fig. 2. DSCOPE prototype on a Raspberry board

In our DSCOPE prototype, the DSCOPE hypervisor consists of 922 SLOC assembly code and 1100 SLOC C code. The DOE manager includes 307 SLOC C code and the UART library has 197 SLOC C code. Since the loader and the UART library run inside the DOE instead of in EL2, the TCB of DSCOPE only has 2022 SLOC C code excluding the bootloaders.

4.1 Hypervisor Implementation

During initialization, the hypervisor recognizes the range of physical memory addresses and sets the register `HCR_EL2` to enable the stage 2 address translation. The hypervisor populates the stage 2 translation tables with one-to-one mapping using the 4KB page size. It reserves 512 KB from `0x0 - 0x80000` for the hypervisor code and 64 MB from `0xF4000000 - 0xF8000000` for the stage 2 translation tables and the DOE software. The reserved memory cannot be accessed by code with EL1 or EL0 privileges. Furthermore, the hypervisor sets up the exception table and handlers for EL2 exceptions and EL2-IRQ so that it can handle the stage-2 page faults and hypcalls from the DOE.

After initializing the basic memory mappings, the hypervisor configures the GIC. First, it sets `HCR_EL2.IMO/FMO` bit to 1, which instructs the hardware to route all non-secure physical interrupts to EL2 for handling. Secondly, it sets the virtual CPU interface control register `GICH_HCR.EN` to 1 to enable vGIC and vCPU interface. It also configures the CPU interface register `GICC_CTLR.EOIMode` to separate End Of Interrupt (EOI) and Deactivate Interrupt (DI) signals. This step prevents interrupts from being blocked because the hypervisor can drop the interrupt priority in GIC via EOI. The base address of GIC CPU interface is remapped to the vCPU interface by the stage 2 translation tables. Lastly, the hypervisor sets register `GICD_IPRIORITYR` to assign the highest priority to GPIO 04 IRQ which is selected as the Activation-IRQ.

4.2 Interrupt Handling

The hypervisor is empowered to intercept, filter and dispatch all external interrupts and software generated interrupts. As shown in Figure 2(b), the hypervisor detects the Activation-IRQ and dispatches other IRQs as vIRQs to the kernel via the GIC. When a physical IRQ occurs, the hypervisor uses the following steps to handle the event:

1. Read the GIC CPU interface register `GICC_IAR` to get IRQ number;
2. Write register `GICC_EOIR` to drop the priority. For SGI/IPI, it writes register `GICC_DIR` to deactivate IRQ;
3. If the IRQ is Activation-IRQ, the manager is to handle it. Otherwise, it writes register `GICH_LR` and sets `HCR_EL2.VI` to 1 so that the IRQ is forwarded as a virtual IRQ to the kernel at the same core;

As a result, the kernel is interrupted and reads `GICC_IAR` which is redirected to `GICV_IAR` to get vIRQ number. After completing handling the vIRQ, the kernel writes `GICC_DIR` which is redirected to `GICV_DIR` to deactivate IRQ.

The physical address of `GICD_SGIR` is protected by the hypervisor so that a kernel's access to it triggers a stage 2 page fault. When active, the hypervisor blocks the access and the kernel cannot deliver an SGI to the DOE core; when inactive, it emulates the access to trigger the SGI.

4.3 Device Operation Environment

The DOE is built on the fly by the hypervisor after receiving the Activation-IRQ. The hypervisor saves the CPU context of the DOE core including the values in register `TTBR0` and `TTBR1`, `SP_EL1` and `ELR_EL2`. It flushes the TLB and invalidate caches for the DOE memory. It builds the stage 1 translation tables for the the local mappings within the DOE and re-configures the stage 2 translation table for memory isolation. Lastly, it switches to EL1 and passes the control to the DOE manager.

Memory Space with Data Consistency All the code and data in the DOE are located in the 4 GB virtual address range from `0x0` to `0x00000000FFFFFFFF`.

The `TTBR0_EL1` register of the DOE core is set with the same value as other cores's `TTBR0_EL1` which points to the root of the stage 1 translation table for the kernel VA (i.e., from `0xFFFFFFFF8000000000` above) translation. As a result, the DOE agent runs in the user-space VA range and can access the kernel's virtual memory since it is granted with the EL1 privilege. This address mapping setup makes kernel memory read/write as native as accessing the agent's own memory.

The isolation between the DOE and the kernel is achieved via the stage 2 address translation. The hypervisor maintains two sets of stage 2 translation tables. One is used for the kernel and the applications on top of it. These tables covers all physical memory frames except those pages reserved for the hypervisor and the DOE. The other set is use for the DOE core and encloses mappings to the DOE memory and identical mappings to the kernel memory.

UART Library The UART library provides several functions for code in the DOE to communicate with trusted external PC. It is loaded at a fixed virtual address. The library has a function that polls register `UART_LSR` (mapped to physical address `0xFE215014`) to acquire the present UART status, as well as read and write functions to access register `UART_LSR` (mapped to physical address `0xFE215010`) for data communications.

DOE Loader The DOE loader is the first piece of code running inside the DOE after the launching. Similar to the UART library, it is loaded at a fixed virtual address. Its main functionality is to communicate with the console at the external PC via the URART channel. Since accessing the kernel memory is trivial, the loader can also introspect the kernel. Specifically, it receives three types of commands from the PC, including importing an incident response agent, introspection, and termination. For the importing command, it accepts and loads the binary image of the agent to the DOE memory. For the introspection command, it receives the starting kernel virtual address and the amount of bytes to read and then returns the fetched memory chunk to the PC.

4.4 DOE Agent

A DOE agent is a raw binary without using the ELF or kernel object format. As the DOE does not provide standard C libraries except the UART library, an agent is required to be self-contained. The entry point of a DOE agent is set to a fixed virtual address defined by `DSCOPE`. The loader starts the agent execution from the default entry point. After the agent is executed, the loader erase agent code and data in the memory.

4.5 Case Studies

To show `DSCOPE`'s capability, we develop two use cases: live kernel object acquisition and CPU status acquisition.

Live Kernel Object Acquisition We implement a kernel introspection agent that parses and dumps kernel objects to the external PC. Since `DSCOPE` does

not force the kernel to stop execution, the agent introspects the kernel memory during kernel execution. The agent is compiled with the definitions of various kernel object structures and those static addresses of kernel symbols. The agent supports traversal of the `task_struct` object list. It lists all process identifiers in the running kernel. In specific, it locates the kernel symbol `init_task` base virtual address to get whole process identifiers and then lists the PID, command, state, stack address and next. For each process, the agent reads 1,776 byte from the starting virtual address of its `task_struct`. A screen snapshot of the agent's output is shown in Figure 3(a).

The agent also dumps all entries of the system call handler table used by the running kernel by accessing the virtual address of `sys_call_table`, which helps the user to check whether they are hooked by a rootkit. To collect I/O related intelligence, the agent lists the device's I/O port status by accessing the memory-mapped status registers of each port and the PIDs of the current interrupt request. These information helps the user to assess whether the device has suspicious I/O operations.

CPU Status Acquisition Although memory acquisition could be performed from the outside using special hardware, CPU status acquisition has to be done within the device. In the event of kernel panic, the kernel panic handler is expected to save all CPU cores' contexts to the kernel log. However, under DSCOPE's threat model, the kernel log could be a faked one. Hence, our second use case of DSCOPE is to collect the genuine CPU contexts.

The basic approach of CPU introspection is to preempt the kernel from its CPU core and trap them to the hypervisor. Initially, only the DOE core is fully controlled by the agent. To introspect another CPU core's context, the agent uses a hypcall to invoke the hypervisor at the DOE core to send SGIs to other cores. As a result, the recipient cores are trapped the hypervisor which responds to the event by saving all general-purpose registers and special registers on EL0/1. Figure 3(b) is a screenshot of the CPU introspection output.

```

C:\uarhandle\Debug\uarhandle.exe
Please enter command:-L

21 bytes wrote.
pid: 0, state: 0x0, cmd: swapper/0, stack: ffffffff011610000, next: ffffffff80bbd88000

21 bytes wrote.
pid: 1, state: 0x200, cmd: systemd, stack: ffffffff010028000, next: ffffffff80bbd89d80

21 bytes wrote.
pid: 2, state: 0x1, cmd: kthreadd, stack: ffffffff010030000, next: ffffffff80bbd8bb00

```

(a) Live acquisition of the running processes from the device

```

fffffc0100000000 ffffffc010000000
CPU introspection start

current_core_id is 00000003
ttbr0 is 000000008548009
sp_el1 is ffffffff010093f00
elr_el2 is ffffffff010080254
general register dump demo
x0 value is 0000000000000020
x1 value is 0000000000000020

End of CPU introspection demo
fffffc010080254 ffffffc010000000
fffffc010093f00 ffffffc010000000
lirq after enable EL1:00000003 0000000E 0000000770FB07
Got TPI from GPT0: 040000003 0000000E

```

(b) Live acquisition of CPU states

Fig. 3. Screenshots from two use cases

5 Evaluation

In this section, we shed light on how DSCOPE resists several types of attacks against its dependability and the reliability of the DOE, followed by various experiments that validate our analysis. We also measure the performance of DSCOPE and report the results.

5.1 Reliability of DScope Launch

The major attack vector is the influence of the kernel adversary on the hypervisor state. The adversary attempts to hinder the hypervisor from entering the active state or to force it to return to the inactive state.

Masking Interrupts Because only the Activation-IRQ triggers DSCOPE to enter the active state, the kernel adversary may try to mask interrupts in order to prevent CPU cores being trapped into the hypervisor. Masking interrupt on all physical cores leads the CPU not to respond to interrupts. Benefit from vGIC virtualizing interrupts, the kernel only can mask EL1-IRQ since masking EL2-IRQ requires the EL2 privilege. In our system design, the external hardware interrupt is treated as an EL2-IRQ and the kernel only handles vIRQs which are virtualized by GIC. Hence kernel interrupt masking cannot prevent the Activation-IRQ from being delivered.

IRQ Flooding The kernel adversary may continuously generate interrupts by reading or writing GPIO registers or IPIs to generate a flood of IRQs. The swarm of IRQs can fill up the IRQ queue in the GIC so that the Activation-IRQ cannot be forwarded to the CPU. DSCOPE counters this attack by leveraging the interrupt priority settings. The hypervisor assigns the Activation-IRQ with the highest priority by setting register `GICD_IPRIORITYR`. Since the interrupts generated by the kernel always have a lower priority, the GIC is ensured to deliver the Activation-IRQ to the CPU interface. Note that the priority register is inaccessible to the kernel. Thus, the IRQ flooding attack cannot block the Activation-IRQ delivery.

Device Restart and Shutdown The kernel adversary may attempt to reboot the device so that the hypervisor is reset to the inactivate state. To deal with the attack, the hypervisor intercepts all SGIs and discards the CPU INIT signals if it is currently in the active state. Besides, the stage 2 translation table protects the registers related to power management to prevent a power off operation.

5.2 Reliability Experiments

The experiments aim at measuring whether DSCOPE can be successfully launched at the presence of a compromised or crashed kernel. We intentionally load several buggy drivers to the kernel running on the Raspberry Pi board and implement five different ways to test DSCOPE's reliability.

Kernel Panic We introduce kernel panic by using the Linux shell command `$echo c sysrq-trigger`, which directly crashes the kernel. We configure the device to restart in 20 seconds after kernel panic and test whether the user can activate DSCOPE within the time frame. The experiment shows that the user can still launch the DOE during kernel panic.

Kernel Crash This experiment tests DSCOPE’s dependability on kernel crash. We change PXN and XN bit to 1 in all kernel page table entries, which makes all kernel code not executable. The DSCOPE activation is not dependent on the kernel because the Activation-IRQ is delivered via a hardware event. The DOE agent does not have dependence on kernel code either. In the experiment, the hypervisor still receives the Activation-IRQ and launches the DOE at a randomly chosen core.

Attack 1. Our injected driver in the kernel disables all I/O interfaces by removing the relevant drivers in the kernel, including the UART, ethernet, Wi-Fi, GPIO and Touchscreen. The goal is to prevent users from interacting with the hypervisor or the device. The experiment shows that the DOE loader can still use the UART to communicate with the PC. DSCOPE has an isolated I/O port that is only accessed by the DOE core.

Attack 2. Our injected driver in the kernel masks interrupts on all CPU cores by setting the DAIF bits. In an ARM processor, these bits mask EL1 interrupts delivered to the core. Because the hypervisor intercepts all IRQs, an EL2-bound interrupt is still delivered to the primary core which is trapped to the hypervisor. The experiment shows that DSCOPE responds as expected when the user taps the two lines shown in Figure 2(a) to trigger the Activation-IRQ.

Attack 3. We modify the kernel so that it stops updating the watchdog on all cores when DSCOPE is active. Failure to update the watchdog on time will cause the device to restart. The hypervisor filters and drops the CPU INIT signal. The experiment shows that the device remains running when DSCOPE in the active state.

In short, the experiment results show that the hypervisor can be successfully launched and function properly in all these experiments, which demonstrates that DSCOPE is resilient against the kernel’s denial of service attacks.

5.3 Security and Reliability of DOE

Next, we analyze how the DOE’s security and reliability is safeguarded against the kernel adversary’s attack.

SGI DoS A direct operation to break core isolation is SGI DoS. The physical cores controlled by the kernel adversary send a flood of SGIs to the DOE core. The aim is to preempt the DOE agent and starve it of the CPU time. As explained in Section 3, the hypervisor filters and drops all IRQs destined to the DOE core after the DOE is set up. Therefore, the SGI DoS attack is unable to disrupt the DOE agent execution.

Break Memory Isolation There are two potential factors that break memory isolation. One is the malicious behavior from the kernel and the other is the programming bug inside the DOE software. DSCOPE realizes two-way memory isolation. On the one hand, the stage 2 translation tables used for the kernel do not have the mappings to physical pages used by the DOE. Thus, the kernel cannot access to the DOE memory. On the other hand, kernel code pages are deprived of execution permissions in the stage 2 translation tables used by the DOR core. Hence, a buggy execution of the DOE code cannot pass the control to the kernel code. Besides, the hypervisor flushes the TLB on the DOE core and invalids caches, which ensures that the DOE software does not use stale data.

5.4 Security Assessment

We run experiments to test the security of DSCOPE against kernel attacks. In the experiments, we insert malicious code into a device driver whose execution can be triggered by using a shell command. The first attack uses the kernel function `cpumask_set_cpu(CPU_ID, &tthread_cpumask)` to stop the DOE core after the DOE is launched. This function masks the selected core by sending IPIs. The experiment shows that the DOE core does not receive the offensive IPI as expected. The hypervisor successfully stops its delivery and logs the IPI request from the isolated I/O port.

To verify one-way read/write access between the DOE and the kernel, our offensive kernel code attempts to access the DOE memory. The experiment reports the stage-2 page faults and traps the CPU into stage-2 exception handler. We instrument stage-2 page fault handler to print out the target physical address in the DOE and the virtual address that generated this page fault. The experimental result validates the memory isolation of the DOE.

5.5 Performance Evaluation

The performance evaluation measures the impact of DSCOPE on the normal use of device as well as the overheads of the hypervisor and the agents in an acquisition session.

Overhead of Interrupts Handling The hypervisor enables interrupt virtualization and changes the workflow of interrupts handling. Compared no virtualization, the overhead of interrupt handling is slightly increased. To measure the overhead, we use the Performance Monitors Cycle Count Register (PMCCNTR) to count the CPU cycles of an interrupt handling. In order to avoid the interference caused by interrupt priority, we set a GPIO IRQ to the highest priority as the test IRQ. With DSCOPE, the time interval from the moment when the hypervisor starts to handle the test IRQ to the moment when the kernel reads the IRQ ID from vCPU interface is about 28,462 cycles in average. Without interrupt virtualization, it takes about 670 cycles for the kernel to access GIC CPU interface and get the IRQ ID. The mainly cost is caused by virtualizing IRQ and the imposed restriction on GIC vCPU interface access.

Overhead of Boot-up and Context Switch We assess the overhead added to device boot up time by measuring the boot-up time between device power-on to displaying the login interface. The average boot up time of the original device is 30.76 seconds. When device is equipped with DSCOPE, the average boot up time grows to 35.29 seconds. The overhead is mainly incurred by configuring the memory virtualization and configuring vGIC.

To measure the context switch time, we use PMCCNTR to measure the time for DSCOPE activation and inactivation. The activation cost consists of the time interval between the Activation-IRQ arrival and sending IPI to the DOE core and the time interval between the DOE receiving this IPI and the agent starts to run. The time cost IPI delivery cannot be measured because each core has its own PMCCNTR. The inactivation cost is between the hypervisor issuing ERET instruction and the the kernel getting the control of the DOE core. Our experiments report that, the first part of activation costs 5,895 CPU cycles and the second part takes 537,435 CPU cycles in average, which adds up to 543,330 cycles. The inactivation costs 730 cycles in average.

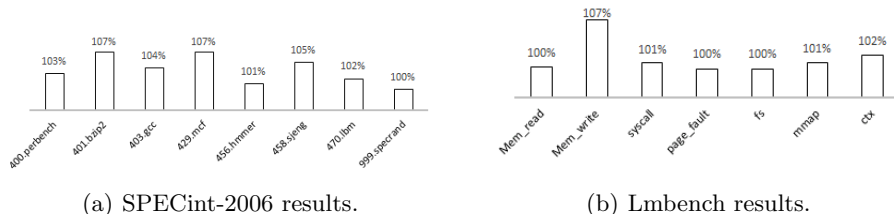


Fig. 4. Performance overhead. The numbers are the percentage of the score with DSCOPE compared to the score without DSCOPE.

Performance Benchmarks We run benchmarks to assess the system performance impact up the device when the hypervisor is inactive. We choose two standard test suits: SPECint-2006 [23] and Lmbench [18]. Each benchmark uses the standard test inputs. We use the Linux command `time` to get the average execution time of the test suite, with and without DSCOPE. The experimental results are reported in Figure 4(a) and Figure 4(b).

The experimental results show that DSCOPE incurs modest performance loss to the device, as the performance drops between 0% to 7%. The main reason of extra overhead on device is due to memory and interrupt virtualization. Compared the system without virtualization, the IPA-to-PA translation and virtualizing interrupts consume more CPU time. We also compare the kernel and the DOE agent’s speed of reading the kernel memory. The TLB is flushed and cache is invalid before each memory experiment. When reading 1024 bytes from the kernel virtual memory, the DOE agent only takes 100 more CPU cycles than the kernel.

6 Conclusion

This paper proposes DSCOPE as a reliable data acquire system on ARM devices. By leveraging ARM virtualization extension and interrupt virtualization, DSCOPE can be activated by a user at any time, even if the device kernel is corrupted by malware. The user can securely acquire various types of volatile data from device with mapping consistency. We implement the DSCOPE prototype on a Raspberry Pi4 development board with an ARM64 processor and demonstrate its strengths in both kernel memory data and CPU register status acquisition. We also measure the reliability of DSCOPE by various experiments including kernel failure and attacks. The experimental results show that DSCOPE is resistant to denial-of-service attacks, easy to deploy, and lightweight in terms of performance overhead.

Acknowledgement. We thank anonymous reviewers and the shepherd for their revision suggestions. This research / project is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme, National Satellite of Excellence in Mobile Systems Security and Cloud Security (NRF2018NCR-NSOE004-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

1. Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Aion: Enabling open systems through strong availability guarantees for enclaves. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1357–1372, 2021.
2. Esmerald Aliaj, Ivan De Oliveira Nunes, and Gene Tsudik. {GAROTA}: Generalized active {Root-Of-Trust} architecture (for tiny embedded devices). In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2243–2260, 2022.
3. ARM. *Generic Interrupt Controller Architecture version 2.0*. 2013.
4. Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. DKSM: Subverting virtual machine introspection for fun and profit. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 82–91. IEEE, 2010.
5. Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Over-shadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLoS)*, 2008.
6. Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. Efficient virtualization-based application protection against untrusted operating system. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.
7. Christoffer Dall and Jason Nieh. KVM/ARM: the design and implementation of the Linux ARM hypervisor. *ACM Sigplan Notices*, 49(4):333–348, 2014.

8. Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (SP)*, pages 297–312. IEEE, 2011.
9. Ulfar Erlingsson, Martin Abadi, Michael Vrabie, Mihai Budiu, and George C Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, 2006.
10. Yangchun Fu and Zhiqiang Lin. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP)*, pages 586–600. IEEE, 2012.
11. Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Fault isolation for device drivers. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 33–42. IEEE, 2009.
12. Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
13. Manuel Huber, Stefan Hristozov, Simon Ott, Vasil Sarafov, and Marcus Peinado. The lazarus effect: Healing compromised devices in the internet of small things. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 6–19, 2020.
14. Joo-Young Hwang, Sang bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *Proceedings of the 5th IEEE Consumer Communications and Networking Conference, CCNC*, January 2008.
15. Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. SoK: Introspections on trust and the semantic gap. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, 2014.
16. Donghyun Kwon, Kuenwhee Oh, Junmo Park, Seungyong Yang, Yeongpil Cho, Brent Byunghoon Kang, and Yunheung Paek. Hypernel: a hardware-assisted framework for kernel protection without nested paging. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.
17. Valentin JM Manes, Daehee Jang, Chanhoo Ryu, and Brent Byunghoon Kang. Domain isolated kernel: A lightweight sandbox for untrusted kernel extensions. *computers & security*, 74:130–143, 2018.
18. Larry McVoy. lmbench: Portable tools for performance analysis. <http://www.bitmover.com/lmbench/>.
19. Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P)*, 2008.
20. Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.
21. Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSES. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

22. Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 477–487. ACM, 2009.
23. SPEC. Standard performance evaluation corporation. <http://www.spec.org/cpu2006/index.html>.
24. Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process out-grafting: an efficient out-of-VM approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 363–374. ACM, 2011.
25. He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. Trustdump: Reliable memory acquisition on smartphones. In *Computer Security-ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I 19*, pages 202–218. Springer, 2014.
26. Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107, 2002.
27. Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2014.
28. Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.
29. Jiang Wang, Fengwei Zhang, Kun Sun, and Angelos Stavrou. Firmware-assisted memory acquisition and analysis tools for digital forensics. In *2011 Sixth IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering*, pages 1–5. IEEE, 2011.
30. Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 352–369. IEEE, 2022.
31. Meng Xu, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom. Dominance as a new trusted computing primitive for the internet of things. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1415–1430. IEEE, 2019.
32. Siqi Zhao and Xuhua Ding. FIMCE: A fully isolated micro-computing environment for multicore systems. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–30, 2018.
33. Siqi Zhao, Xuhua Ding, Wen Xu, and Dawu Gu. Seeing through the same lens: Introspecting guest address space at native speed. In *Proceedings of the 26th USENIX Security Symposium*, 2017.
34. Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMLock: Hardware-based fault isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 558–569, 2014.