HART: Hardware-Assisted Kernel Module Tracing on Arm

Yunlan Du[1], Zhenyu Ning[2], Jun Xu[3], Zhilong Wang[4], Yueh-Hsun Lin[5], Fengwei Zhang[2](✉), Xinyu Xing[4], and Bing Mao[1]

[1] Nanjing University, Nanjing, China
[2] Southern University of Science and Technology, Shenzhen, China
zhangfw@sustech.edu.cn
[3] Stevens Institute of Technology, Hoboken, USA
[4] Pennsylvania State University, State College, USA
[5] JD Silicon Valley R&D Center, Mountain View, USA

Abstract. While the usage of kernel modules has become more prevalent from mobile to IoT devices, it poses an increased threat to computer systems since the modules enjoy high privileges as the main kernel but lack the matching robustness and security. In this work, we propose HART, a modular and dynamic tracing framework enabled by the Embedded Trace Macrocell (ETM) debugging feature in Arm processors. Powered by even the minimum supports of ETM, HART can trace binary-only modules without any modification to the main kernel efficiently, and plug and play on any module at any time. Besides, HART provides convenient interfaces for users to further build tracing-based security solutions, such as the modular AddressSanitizer HASAN we demonstrated. Our evaluation shows that HART and HASAN incur the average overhead of **5%** and **6%** on **6** widely-used benchmarks, and HASAN detects all vulnerabilities in various types, proving their efficiency and effectiveness.

Keywords: Kernel module · Dynamic tracing · ETM · Arm

1 Introduction

To enhance the extensibility and maintainability of the kernel, Linux allows third parties to design their own functions (*e.g.,* hardware drivers) to access kernel with loadable kernel modules. Unlike the main kernels developed by experienced experts with high code quality, the kernel modules developed by third parties lack the code correctness and testing rigorousness that happens with the core, resulting in more security flaws lying in the kernel modules. Some kernel vulnerability analyses [25,40] show that CVE patches to the Linux repository involving kernel drivers comprise roughly 19% of commits from 2005 to 2017. The conditions only worsen in complex ecosystems like Android and smart IoT devices. In

Y. Du and Z. Ning—These authors contributed equally to this work.

© Springer Nature Switzerland AG 2020
L. Chen et al. (Eds.): ESORICS 2020, LNCS 12308, pp. 316–337, 2020.
https://doi.org/10.1007/978-3-030-58951-6_16

2017, a study on vulnerabilities in the Android ecosystem [39] shows that 41% of 660 collected bugs came from kernel components, most of which were device drivers.

To mitigate the threats of vulnerable kernel modules, the literature brings three categories of solutions. As summarized in Table 1, all these solutions carry conditions that limit their use in practice. The first category of solution is to detect illegal memory access [5,6,11,16] at run-time, aiming to achieve memory protection in the kernel. As such solutions require to instrument memory accesses and perform execution-time validation, they need source code and incur high performance overhead. These properties limit solutions in this category for only debugging and testing. The second category of approaches explores to ensure the integrity of the kernel [23,26,53,60,64], including control flow, data flow and code integrity. However, these strategies introduce significant computation overhead and often require extensive modification to the main kernel. The last category of solutions isolates the modules from the core components [22,24,31, 48,51,54,58,61,65], so as to confine the potentially corrupted drivers running out of the kernel. By design, the isolation often requires source code and comes with significant instrumentation to the kernel and the driver.

Table 1. Comparison between existing kernel protection works and ours. *Non-intrusive* represents whether the tools have system-level modification of kernel, user space libraries, etc. (✓ = yes, ✗ = no, ✱ = partially supported).

Approach category	Binary-support	Non-intrusive	Low overhead	Representative works (in the order of time)
Memory Debugger	✗	✗	✗	Slub_debug [5], Kmemleak [16], Kmemcheck [11], KASAN [6]
Integrity Protection	✗	✗	✱	KOP [23], HyperSafe [60], HUKO [64], KCoFI [26], DFI for kernel [53]
Kernel Isolation	✗	✗	✱	Nooks [54], SUD [24], Livewire [22], SafeDrive [65], SecVisor [51]
Our method	✓	✓	✓	HASAN

To overcome these limitations, we propose HART, a high performance tracing framework on Arm. The motivation of HART is to dynamically monitor the execution of the third-party Linux kernel modules even without module source code. HART utilizes the ETM debugging component [9] for light-weight tracing of both control flow and data access. Combing hardware configurations and software hooks, HART further restricts the tracing to selectively work on specific module(s). With the support of the open interfaces provided by HART, various security solutions against module vulnerabilities can be established. We

then demonstrate a modular AddressSanitizer named HASAN. Compared with the previous solutions, our approach is much less intrusive as it requires zero modification to other kernel components, and is fully compatible with any commercial kernel module since it requires no access to any source code. Besides, our empirical evaluation shows that HASAN can detect the occurrence of memory corruptions in binary-only modules with a negligible performance overhead.

The design of HART overcomes several barriers. First, for generality, we need to support ETM with the minimum configuration, particularly the size of Embedded Trace Buffer (ETB) that stores the trace. Take Freescale i.MX53 Quick Start Board as an example. Its ETB has a size of 4 KB and can get fully occupied very frequently. The second barrier is that many implementations of ETB raise no interrupt when the buffer is full, thereby the trace can be frequently overwritten. To tackle the above two challenges and retrieve the entire trace successfully, we leverage the Performance Monitoring Unit (PMU) to throw an interrupt after the CPU executes a certain number of instructions, ensuring that the trace size will never exceed the capacity of ETB. Finally, to optimize the performance of HART, we design an elastic scheduling to assure the decoding thread of rapid parsing while avoiding the waste of CPU cycles. Under this scheduling, the thread will occupy the CPU longer while the trace grows faster. Otherwise, the thread will yield the control of CPU more frequently.

Our main contributions are summarized as follows.

- We design HART, an ETM-assisted tracing framework for kernel modules. With the minimal hardware requirements of ETM, it achieves low intrusiveness, high efficiency, and binary-compatibility for kernel module protections.
- We implement HART with Freescale i.MX53 Quick Start Board running Linux 32-bit systems. Our evaluation with **6** widely used module benchmarks shows that HART incurs an average overhead of **5%**.
- We build HASAN on the top of HART to detect memory vulnerabilities triggered in the target module(s). We evaluate HASAN on **6** existing vulnerabilities ported from real-world CVE cases and the above benchmarks. All the vulnerabilities are detected with an average overhead of only **6%**.

2 Background and Problem Scope

In this section, we first introduce the background of kernel module and ETM, and then define our problem scope.

2.1 Loadable Kernel Module

In practice, the third-party modules are mostly developed as loadable kernel modules (LKM). An LKM is an object file that contains code to extend the main kernel, such as building support for new hardware, installing new file systems, etc. For the ease of practical use, an LKM can be loaded at anytime on-demand and unloaded once the functionality is no longer required. To support accesses

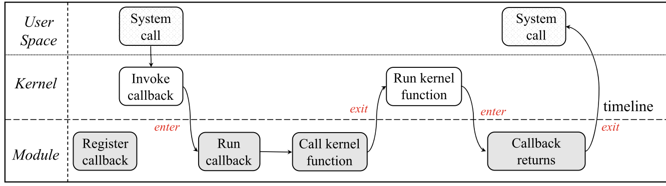

Fig. 1. Interactions between an LKM and the users space as well as kernel space.

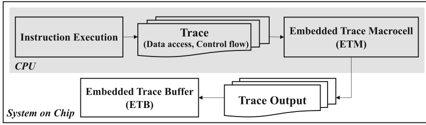

Fig. 2. A general model of ETM.

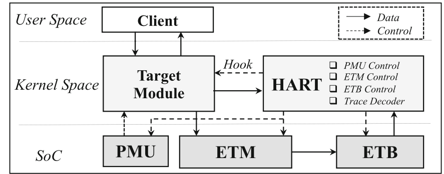

Fig. 3. Architecture of HART.

from the user space, LKM usually registers a group of its functions as callbacks to system calls in the kernel (*e.g.,* `ioctl`). In Fig. 1, a client (*e.g.,* user space application) can execute those callbacks via the system calls. Sitting in the kernel space, an LKM can also freely invoke functions exported by the main kernel.

2.2 Embedded Tracing Microcell

ETM is a hardware debugging feature in Arm processors since two decades ago. It works by tracing the control flow and memory accesses of a software's execution. Being aided by hardware, ETM incurs less than 0.1% performance overhead [43], barely decelerating the normal execution. While ETMs on different chips have varied properties, it generally follows the model in Fig. 2. When CPU executes instructions, ETM collects information about control flows and memory accesses. Such information can be compressed and stored into an on-chip memory called Embedded Trace Buffer (ETB), which is accessible through mapped I/O or debugging interfaces such as JTAG or Serial Wire interface.

Trace generated by ETM follows standard formats. A `Sync` packet is produced at the start of execution, containing information about the entry point and the process ID (`CID`); a `P-header` packet indicates the number of sequential instructions and not-taken instructions executed since the last packet; a `Data` packet records the memory address to be accessed by the instruction. We can fully reconstruct the control flow and memory accesses by decoding such packets.

2.3 Problem Scope

We consider the generality of our work from two aspects. On the one hand, the framework should have compatibility with the majority (if not all) of ETM. We investigate the ETM features on several early and low-end SoCs in Table 2,

Table 2. ETM and ETB on early SoCs.

SoC	Devices	ETB size	ETM feature	
			DAT[a]	ARF[b]
Qualcomm Snapdragon 200 [18]	Xperia E1, Moto E	≥4 KB	×	✓
Samsung Exynos 3110 [17]	Galaxy S, Nexus S	≥4 KB	✓	✓
Apple A4 [7]	iPhone 4	≥4 KB	✓	✓
Huawei Kirin 920 [36]	Mate 7, Honor 6	≥4 KB	×	✓
NXP i.MX53 [49]	iMX53 Quick Start Board	4 KB	✓	✓
Arm Juno [4]	Juno r0 Board	64 KB	×	✓

a: Data Address Trace
b: Address Range Filter

and summarize the following most preliminary ETM configuration for HART to work with: ① ETM supports tracing of both control flow and memory accesses. ② ETM supports filtering by address range. ③ ETB is available for trace storage, yet with a limited buffer size - 4 KB. ④ ETB raises no interrupts when it gets full.

On the other hand, the framework should not impose restrictions on the target modules. We then accordingly make a minimal group of assumptions about the target module: ① Source code of the target module is unavailable. ② The target module and other kernel components shall be intact without any modification. ③ The target module is not intentionally malicious.

3 HART Design and Implementation

In this section, we concentrate on HART, our selective tracing framework for kernel modules. We first give an overview of HART in Fig. 3. HART is designed as a standalone kernel driver. At the high-level, it manages ETM (and ETB) to trace the execution of target modules, and PMU to raise interrupts to address the problem of overwritten trace in ETB, and a Trace Decoder to parse the ETM trace. HART also hooks entrances/exits to/from the target modules, to coordinate with PMU and provide open interfaces for further usage. Following the workflow, we then delve into the details of our design and implementation.

3.1 HART Design

Initialization for Tracing. Before starting a module, the Linux kernel performs a set of initialization, including loading data and code from the module, relocating the references to external and internal symbols, and running the `init` function of the module. HART intercepts the initialization to prepare for tracing. Details are as follows.

First, HART builds a profile for the module, with some necessary information about the loading address, the `init` function, and the kernel data structure that manages this module, for later usage. Since such information will be available

after the kernel has finished loading the module, we capture the initialization at this point of time with a callback registered through the trace-point in the `load_module` kernel function, without intrusion to the kernel.

Then, HART hooks the entrances to the module and the exits from the module, which is the base of PMU management and open interfaces for users of HART. As Fig. 1 depicted before, the modules are mainly entered and exited either through the callbacks registered to the kernel, or the external functions invoked by such callbacks. We deal with different cases as follows.

In the first case, those callbacks are typically functions within the module, waiting for the invoking from the kernel. They are usually registered through code pointers stored in the `.data` segment. In the course of module loading, the code pointers will be relocated to the run-time addresses during the initialization, so that we can intercept the relocation and alter these pointers to target the wrappers defined by HART. The comparison before and after the HART's alteration in the code pointers' relocation can be observed from Fig. 4a, presented with concrete examples. Thus, our wrappers are successfully registered as the callbacks to be invoked by the kernel, and can perform a series of HART-designed handlers, while ensuring the normal usage of original functions.

In the other case that callbacks invoke external functions, such external symbols (*e.g.,* `kmalloc`) also need a similar relocation to fix their references to the run-time addresses. Likewise, HART intercepts such relocation and directs the external symbols to the HART-designed wrappers, as is shown in Fig. 4b. So far, as we consider both conditions, all of the entrances and exits are under the control of HART via the adjusted relocation to our wrappers.

Finally, HART gets ETM and ETB online for tracing. To be specific, HART allocates a large continuous buffer `hart_buf`, for the sake of backing up ETB data. In our design, its buffer size is configured as `4 MB`. Meanwhile, HART spawns a child thread, which monitors data in `hart_buf` and does continuous decoding. We will cover more details about the decoding later in Sect. 3.1. Then, HART enables ETM with ETB, and configures ETM to only trace the range that stores the module. So far, HART finishes the initialization and starts the module by invoking the module's `init` function.

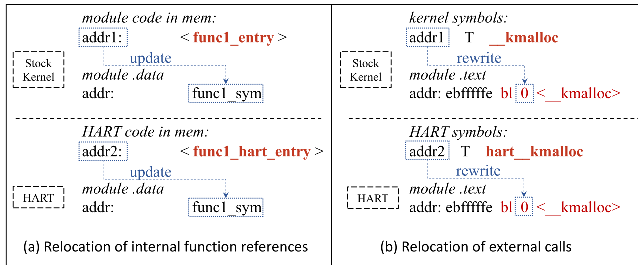

Fig. 4. Relocation of external calls and internal function references in modules.

Table 3. Wrapper for `func1` in Fig. 4a.

```
1  ENTRY(func1_hart):
2      SAVE_CTX //save context
3      MOV R0,LR //pass and save LR in HART
4      BL resume_pmu //call resume_pmu
5      MOV LR,R0 //return addr of ori. func1
6      RESTORE_CTX //restore context
7      BL LR //call original func1
8      SAVE_CTX //save context
9      BL stop_pmu //call stop_pmu
10     MOV LR,R0 //return LR from HART
11     RESTORE_CTX //restore context
12     BL LR //return back
```

Table 4. Wrapper for `__kmalloc`.

```
1  void * hart__kmalloc(size_t size,
2                       gfp_t flags){
3    /*stop PMU*/
4    cur_cnt = stop_pmu();
5    /*instrumentation hooks*/
6    pre_instrumention();
7    /*original kmalloc*/
8    addr = __kmalloc(size, flags);
9    /*instrumentation hooks*/
10   post_instrumentation();
11   /*resume PMU*/
12   reset_pmu(cur_cnt);
13   return addr;
14 }
```

Continuous and Selective Tracing. After the above initialization, ETM will trace execution of the module and store its data in ETB. Under the assumption that the ETB hardware raises no interrupt to alert HART to a full ETB, the remaining challenge is to timely interrupt the module before ETB is filled up. In this work, HART leverages the instruction counter shipped with PMU.

Specifically, we start PMU on counting the instructions executed by the CPU. Once the number of instructions hits a threshold, PMU will raise an interrupt, during which HART can copy ETB data out to `hart_buf`. Generally, an Arm instruction can lead to at most `6` bytes of trace data[1]. This means a 4 KB ETB can support the execution of at least `680` instructions. Also considering that the PMU interrupts often come with a skid (less than 10 instructions [43]), we set up `680-10` as the threshold for PMU. In our evaluation with real-world benchmarks, we observe the threshold is consistently safe for avoiding overflow in ETB.

Though PMU aids HART in preventing trace loss, it causes an extra issue. PMU counts every instruction executed on the CPU, regardless of the context. This means PMU will interrupt not only the target module, but also the other kernel components, leading to high-frequency interruptions and tremendous down-gradation in the entire system. To restrict PMU to count inside the selective module(s), we rely on the hooks HART plants during initialization. When the kernel enters the module via a callback, the execution will first enter our wrapper function. In the wrapper, we resume PMU to count instructions, invoke the original callback, and stop PMU after the callback returns in order, as is shown in Table 3. Our wrapper avoids any modification to the stack and preserves all needed registers when stop/resume PMU, so as to keep the context to call the original `init` and maintain the context created by `init` back to the kernel. As aforementioned, the target module may also call for external functions, which are also hooked. In those hooks, HART first stops PMU, then invokes the

[1] Some instructions (*e.g.,* `LDM`) carry more than `6` bytes of trace data, but appear in a relatively few cases. We also take into account the data generated by data access in ETB. Actually, the information in the raw trace output is highly compressed, and a single byte in the trace output can represent up to 16 instructions in some cases.

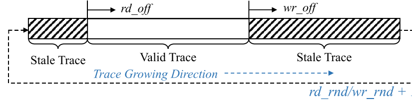

Fig. 5. Parallel trace data saving and decoding.

intended external function, and finally resumes PMU. Table 4 demonstrates the wrapper function of `__kmalloc` when it is called by the target module.

The last challenge derives from handling PMU interrupts. By intuition, we can simply register a handler to the kernel, which stops PMU, backs up ETB data, and then resumes PMU. However, this idea barely works in practice. Actually, on occurrence of an interrupt, the handling starts with a general interface `tzic_handle_irq`. This function retrieves the interrupt number, identifies the handler registered by users, and finally invokes that handler. After the user handler returns, `tzic_handle_irq` will check for further interrupts and handle them. Oftentimes, when we handle our PMU interrupts, new interrupts would come. If we resume PMU inside our handler, PMU would count the operations of `tzic_handle_irq` handling the newly arrived interrupts. Following the `tzic_handle_irq` handling process, it would trigger another PMU interrupt handler, eventually resulting in endless interrupt handling loop.

To address the above problem, we hook the `tzic_handle_irq` function with `hart_handle_irq`. And right before it exits to the target module, we resume PMU. When our PMU interrupts come, the handler `hart_pmu_handler` regis-

Table 5. Algorithm for backing up ETB data inside PMU interrupt handler and the parallel decoding thread for parsing trace with an elastic scheduling scheme.

```
1  void retrieve_trace_data(){
2    /*get trace size*/
3     trace_sz = bytes_in_etb();
4     /*copy trace*/
5     cpy_trace(trace_buf, etb, trace_sz);
6     /*update write offset and rounds*/
7     wr_off += trace_sz;
8     if(wr_off >= trace_buf_sz){
9         wr_off %= trace_buf_sz;
10        wr_rnd ++;
11    }
12    /*overwrite is possible,
13    pause the target module*/
14    if(wr_rnd > rd_rnd &&
15        rd_off - wr_off < 4KB)
16        SIGSTP(target_pid);
17 }
```

(a) ETB data backup.

```
1  void parse_trace(){
2    while(true){
3      /*calculate valid trace size*/
4      if(rd_rnd == wr_rnd) trace_sz =
5        wr_off - rd_off;
6      else trace_sz = trace_buf_sz -
7        rd_off + wr_off;
8      /*decode trace*/
9      decode(trace_buf, rd_off, trace_sz);
10     /*update rd_off and rd_rnd*/
11     rd_off = wr_off;
12     if(rd_rnd < wr_rnd)
13     rd_rnd ++;
14     if(target_paused(target_pid))
15     SIGCONT(target_pid)
16     /*yield CPU based on the
17     workload of decoding*/
18     msleep(100 - CONS *
19       trace_sz/trace_buf_sz);
20     yield();
21   }
22 }
```

(b) Parallel decoding thread.

tered by us will back up the ETB data. In this way, we prevent the above endless interrupt handling loop while obtaining continuous and selective ETM trace.

Elastic Decoding. With the help of PMU, HART can timely interrupt the target module and back up the ETB data. To reduce time cost of backup, in the interrupt handler, HART simply copies the ETB data to the previously allocated `hart_buf`. Meanwhile, HART maintains `wr_off` and `wr_rnd`, respectively indicating the writing position to `hart_buf` and how many rounds HART has filled up this buffer. To decode the trace efficiently, HART spawns a decoding thread for each target module. This thread monitors and keeps decoding the remaining trace data, where `rd_off` and `rd_rnd` indicate the current parsing position and how many rounds it has read through the entire buffer. This design of parallel trace backup and decoding, as shown in Fig. 5, however, involves two issues. We describe the details and explain how we address them as follows.

First, an over-writing problem is likely to happen when the backup function runs beyond the decoder by more than one lap, leading to the error of losing valid data. A similar over-reading error comes due to a similar lack of synchronization. In order to maintain the correctness of the concurrency, we design algorithms for both the backup function and the decoder with checks of over-writing/reading possibilities. On the one hand, as the backup algorithm in Table 5a presents, if `wr_rnd` is larger than `rd_rnd` while `rd_off` is less than `4 KB` ahead of `wr_off`, a new operation of backing up `4 KB` data will potentially overwrite the previous unprocessed trace. In case of this, HART will pause the task that is using the target module, and restart via the decoder later when it is safe to continue. On the other hand, as the decoding algorithm in Table 5b depicts, if the decoder and the backup function arrive at the same location, the decoding thread will pause and yield the CPU due to no trace left to parse.

Second, since the generation of trace is oftentimes slow, a persistent query for any new trace will bring CPU with a waste of cycles. Such frequent occupation of CPU has a negative impact on other normal works, incurring heavy overhead in the performance and efficiency especially for the system equipped with fewer CPU cores. Thus, we introduce an elastic scheduling scheme to the decoding thread as a remedy for the waste, shown in Table 5. During a round of checking and decoding new trace, we calculate the size of the valid trace to represent the generation rate of the trace. Then, the sleep time for the decoding thread is set to inversely proportional to the computed growth rate. That is, the slower the trace grows, the longer the thread sleeps, and vice versa. In this way, HART leaves more resources for CPU while the decoding thread is not busy, but also assures the decoder of high efficiency when it owns a heavy workload.

Supports of Concurrency. We consider two types of concurrency, and support both conditions. First, multiple modules might be registered for protection under HART at the same time. In our design, the context of each module is maintained separately, and HART switches the context at module switch time.

The enter and exit events of a kernel module are captured by HART, and context of ETM tracing and PMU counting related to the module is saved at exiting and restored at reentering. Since traces pertaining to different modules are distinguishable based on the context-ID[2] packets, our design reuses the same `hart_buf` for different modules for the sake of memory efficiency.

Second, multiple user space clients may access the module(s) concurrently, which is common on multi-core SoCs. To handle this type of concurrency, we allocate multiple `hart_buf` on a device with many cores, with each of `hart_buf` uniquely serving one CPU core. This prevents race on trace storage among CPU cores. As both ETM and PMU are core-specific, their configurations and uses are independent across different cores. It is also worth noting that the ETB is shared by the cores. This does not interrupt our system. First, ETM traces from different cores are sequentially pushed to ETB, avoiding race on trace saving. Second, ETM attaches a context-ID whenever a task is being traced. This enables the decoder to separate the trace for individual tasks. Finally, we signal to pause the target module(s) on other cores while one core is doing ETB backup, so as to avoid losing trace data. Other than the trace buffer, we assign a separated decoding thread for each module to prevent confusion in the decoding process.

Open Interfaces. To facilitate the further establishment of various trace-based security applications, we provide users of HART with a comprehensive set of interfaces, including (1) **Initialization and Exiting interfaces** to help users register to the initialization and exiting process of HART, (2) **Decoding interfaces** to pass the parsed ETM trace data to users, and (3) **Instrumentation interfaces** to help hook external functions as shown in line `6` and `10` in Table 4.

3.2 HART **Implementation**

We implement the prototype of HART on Linux system with kernel 4.4.145-armv7-x16, running on a Freescale i.MX53 Quick Start Board [14] with ARMv7 Cortex-A8 processor [20]. As the processor only supports 32-bit mode, HART so far only runs with 32-bit systems. We believe there is no significant difference between the 32-bit and 64-bit implementations, and the only differences might lie in the driver for the involved hardware (*e.g.,* PMU, ETM, and ETB) and some configurations (*e.g.,* the instruction threshold for the trace buffer). In total, the HART system contains about `3,200` lines of `C` code.

To decode the trace data, we reuse an open-source decoder [57], which supports both ETM-V2 and ETM-V3. We optimize the decoding process by directly indexing the packet handler with the packet type. Thus HART avoids the cost of handler looking up and will achieve better efficiency.

After loading HART into the kernel as a standalone driver, we need to customize the native module utilities, `insmod` and `rmmod`, to inform HART of the installation and the removal of the target module to be protected. Before

[2] The context-ID of a task is identical to the process ID. With the context-ID, the processor and ETM can identify which task triggers the execution of the module.

installing the module, our insmod will send the notice to the virtual device registered by HART, which actually reminds HART of standby for the module protection. Our rmmod is customized similarly.

4 HASAN Design and Implementation

We also build an AddressSanitizer HASAN with HART's open interfaces to demonstrate the extended utilization of HART. The design of HASAN reuses the scheme of AddressSanitizer [50] and follows the implementation of the Kernel Address Sanitizer (KASAN) [15]. Instead of sanitizing the whole kernel, HASAN focuses on specific kernel modules. Without source code, we cannot locate local and global arrays, which prevents us from wrapping those objects with redzones. Thus, HASAN only focuses on memory objects on the heap. Note that if the module source code is available, HASAN can achieve the same level protection with KASAN. However, if the module source code is out of reach, HASAN can still achieve heap protection while KASAN will not work at all.

Table 6. Memory management interfaces HASAN hooked.

Category	Allocation	De-allocation
Kmem_cache	kmem_cache_alloc kmem_cache_create	kmem_cache_free kmem_cache_destroy
Kmalloc	kmalloc krealloc kzalloc kcalloc	kfree
Page operations	alloc_pages __get_free_pages	free_pages __free_pages

Design and Implementation: As HASAN aims to protect kernel memory, it only allocates shadow memory for the kernel space. As the kernel space ranges from 0xbf000000 to 0xffffffff in our current system, HASAN allocates a 130 MB continuous virtual space for the shadow memory with each shadow memory bit mapping to a byte. Given an address addr, HASAN maps it to the shadow memory location using the following function, where ADDR_OF_SHM indicates the beginning of our virtual space.

```
1   void * addr_to_hasanaddr(addr){
2       return ((u32)addr >> 3) +
3           (addr - 0xbf000000) + ADDR_OF_SHM);
4   }
```

To wrap heap objects with redzones, HASAN hooks the slab interfaces for memory management[3]. As aforementioned, we achieve the hooking through the interfaces that HART plants to the wrappers of external calls. The interfaces that HASAN currently supports are listed in three categories in Table 6, with their handling methods explained as follows.

① When kmem_cache_create is invoked to creates a memory cache with *size* bytes, we first enlarge this *size* so as to reserve space for redzones around the

[3] We are yet to add supports for slub.

actual *object*. After `kmem_cache_alloc` has allocated the *object*, we mark the rest space as redzones. If `kmem_cache_alloc` requires memory from caches created by other kernel components (in many cases it will happen), HASAN will redirect the allocation to caches that are pre-configured in HASAN. These pre-configured caches will be destroyed when the target module is removed.

② In the second category, we first align the requested size by `8` bytes and then add another `16` bytes reserving for redzones. Then we call stock `kmalloc` and return to users with the address of the `9th` byte in the allocated buffer. The first `8` bytes together with the remaining space at the end will be poisoned.

De-allocations in both categories above follow similar strategies to KASAN. HASAN delays such operations (*e.g.,* `kfree`) and marks the freed regions as poisoned. These memories will be released until the size of delayed memory reaches a threshold or the module is unloaded.

③ In the third category, however, we only delay the corresponding de-allocation. For allocation, increasing the allocated *size* in those interfaces incurs a tremendous cost, as the *size* in their request indicates the order of pages to be allocated.

Table 7. Utility kernel functions for memory operations.

Category	Function name	Category	Function name
memory	`memcpy memccpy memmove memset`	print	`sprintf vsprintf`
string	`strcpy strncpy strcat strncat strdup`	kernel	`copy_from_user copy_to_user`

Since our design excludes the main kernel for tracing, it may miss to capture some vulnerable memory operations inside the kernel functions that listed in Table 7. To handle this issue, we make copies of those functions and load them as a new code section in HASAN. When a target module invokes those functions, we redirect the execution to our copies, enable ETM to trace on them and integrate this trace into ETB. To avoid switching hardware configurations on entering and leaving those functions, our implementation adds the memory holding our copies as an additional range to trace.

The detection of HASAN is straightforward and efficient. We register a call back to HART's decoder. On the arrival of a `normal_data` packet, the decoder will parse the memory address for HASAN. Then, HASAN maps the address to the shadow memory and checks the poison in it. To avoid unsynchronized cases such as decoding of valid accesses after memory de-allocation, we ensure that the decoding synchronizes with the execution in each interface as presented in Table 6.

Fine-Grained Synchronization: As our decoding proceeds after the execution, there is a delay from the occurrence of memory corruption to the detection of violations, which may be exploited to bypass HASAN. To reduce this attack surface, we force the decoder to synchronize with the execution whenever a

callback is invoked. Specifically, our wrapper of that callback will pause the execution until the decoder consumes all the trace. Since a system call reaching the target module will invoke at least one callback, we ensure one or more synchronization(s) between successive system calls. This follows the existing research [21] that relies on system-call level synchronization to ensure security. Another rationale is that, even the state-of-art kernel exploits would require multiple system calls to compromise the execution [62,63].

5 Evaluation

In this section, to demonstrate the utility of our work, we sequentially test whether our work can achieve continuous tracing and measure their efficiency and effectiveness.

5.1 Setup

To understand the correctness and efficiency of our work, we run `6` widely-used kernel modules and come with standard benchmarks, detailing in Table 8. To further understand the non-intrusiveness in our work, we run the `lmbench` benchmarks. The evaluation is conducted on the aforementioned Freescale i.MX53 Quick Start Board [14] with 1 GB of RAM. As we aim to demonstrate the generality of HART with the minimal hardware requirements of the trace functionality of ETM and a small ETB, we consider i.MX53 QSB a perfect match. We do not measure the two types of concurrency described in Sect. 3.1, because the device has only one core, so concurrent tests would produce same results as running the tests sequentially. For comparison, we also test under the state-of-art solution `KASAN` [6]. Since the i.MX53 board we use can only support 32-bit systems while `KASAN` is only available on 64-bit systems, we run the `KASAN` based experiments in a `Raspberry Pi 3+` configured with the same computing power and memory.

To verify the effectiveness of detecting memory corruption, we collect **6** known memory vulnerabilities which cover different types and lie in different modules with various categories and complexities. Note we only cover heap-related buffer overflows because HASAN focuses on heap related memory errors.

5.2 Continuous Tracing

In Table 9, we summarize the frequency of HART backing up the data from the ETB, the size of data retrieved each time. Most importantly, the last column in Table 9 shows ETB has never been filled up. Besides, the maximal size of trace is uniformly less than 2K, not even reaching the middle of ETB. Both facts indicate that overflow in ETB never happened and all the trace data has been successfully backed-up and decoded. Thus, HART and HASAN achieve continuous tracing, which is the precondition for the correctness of the solution.

Table 8. Performance evaluation. The results are normalized using the tests on *Native img + Native module* as baseline 1. A larger number indicates a larger data transmission rate and a lower deceleration on performance.

Module		Benchmark		Result			
Type	*Name*	*Name*	*Setting*	*Native img +*		*KASAN img +*	
				HART *module*	HASAN *module*	*Native module*	*KASAN module*
Network	HSTCP [30]	iperf [29]	Local Comm.	1.00	1.00	0.29	0.28
	TCPW [2]	iperf [29]	Local Comm.	0.92	0.91	0.28	0.28
	H-TCP [12]	iperf [29]	Local Comm.	0.94	0.94	0.26	0.25
File system	HFS+ [13]	IOZONE [28]	Wr/fs=4048K/reclen=64	1.00	1.00	0.96	0.95
			Wr/fs=4048K/reclen=512	0.88	0.87	0.96	0.94
			Rd/fs=4048K/reclen=64	0.92	0.89	0.98	0.92
			Rd/fs=4048K/reclen=512	0.90	0.89	0.99	0.99
	UDF [19]	IOZONE [28]	Wr/fs=4048K/reclen=64	0.95	0.93	0.99	0.97
			Wr/fs=4048K/reclen=512	0.97	0.97	1.00	0.92
			Rd/fs=4048K/reclen=64	0.98	0.97	0.99	0.98
			Rd/fs=4048K/reclen=512	0.97	0.96	1.00	0.98
Driver	USB_STORAGE [8]	dd [45]	Wr/bs=1M/count=1024	1.00	1.00	1.00	0.43
			Wr/bs=4M/count=256	1.00	1.00	0.99	0.43
			Rd/bs=1M/count=1024	0.99	0.99	0.99	0.75
			Rd/bs=4M/count=256	1.00	1.00	1.00	0.76
Avg.	–	–	–	0.95	0.94	0.85	0.72

Table 9. Tracing evaluation of HART and HASAN.

Module		Retrieving times		Max size(Byte)		Min size(Byte)		Average size(Byte)		Full ETB	
Type	Name	HART	HASAN	HART	HASAN	HART	HASAN	HART	HASAN	HART	HASAN
Network	HSTCP	4243	3964	1100	1196	20	20	988	1056	0	0
	TCP-W	3728	3584	1460	1456	20	20	1128	1088	0	0
	H-TCP	3577	3595	1292	1304	20	20	1176	1168	0	0
File system	HFS+	30505	30278	1652	1756	20	20	144	148	0	0
	UDF	17360	20899	2424	2848	20	20	240	232	0	0
Driver	USB_STORAGE	9316	9325	1544	1692	20	20	448	448	0	0

We also observe that HART and HASAN have some similarities and differences in their tracing behaviours. Overall, HART and HASAN present the similar but non-identical results in the same set of module. The slight deviation mainly derives from the padding packets randomly inserted by ETM and the random skids of PMU. The differences lie across different families of modules. HART and HASAN back up ETB more frequently in the two file systems (nearly `2X` more than that in the driver module and `3-7X` more than the network modules), while they produce the least trace in each ETB backup with

the file systems. This indicates that file systems execute more instructions yet they usually carry less information about control flow and data accesses. Due to such increase in instruction number, PMU has to raise more interrupts in the testing on file systems and contributes to the more frequent ETB backups. The statistics above are consistent with the following performance evaluation – a higher frequency of backing up incurs more trace management and thus, introduces higher performance overhead.

5.3 Performance Evaluation

For performance measurement, we record (1) the bandwidths of the server for network modules and (2) the read/write rates for file-systems as well as driver modules. The testing results are summarized in Table 8. Across all three types of drivers, HART and HASAN barely affects the performance. On average, HART introduces an overhead of **5%**, while HASAN introduces **6%**. In the worst-case scenario, HART brings a maximum of 12% for (reclen=64) when testing on HFS+, and HASAN brings at most 13% in the same case. These results well support that HART and HASAN are in general efficient. The most important reason for this efficiency, we believe, is caused by employing ETM to capture control flow and data flow with high efficiency. In general, the hardware-based tracing involves negligible performance overhead, and almost all the overhead introduced by HART is caused by decoding and analyzing the trace output.

On contrast, since KASAN cannot be deployed to the modules without KASAN-enabled kernel, we analyze the impacts of only enforcing KASAN through the differences between the tests with and without KASAN-enabled modules to some extent: ① The test performed on the network modules indicates KASAN introduces nearly 4X slowdown. The reason is that, during a networking event (receiving and sending a message), the kernel performs large amounts of computations due to the complicated operations on the network stack, even though the IO through the network port is cheap as we are only doing local communication. As such, the overhead introduced by KASAN is not diluted by IO accesses. Furthermore, though the difference between two tests on KASAN is insignificant, it does *not* imply that KASAN will be efficient if we re-engineer it to protect the target module only. In a network activity, those drivers only take up a small fraction of computation in the kernel space. Overhead on such computation is covered up by the significant slowdown in the main kernel. ② In the two file systems, KASAN hardly affects the performance of the kernel or the target module. Again, this does *not* necessarily imply that KASAN is most efficient, because in a file system operation, most of the time is spent on disk access and the computation in the kernel is quite simple and quick. Therefore, the average overhead is insignificant. ③ In the test on USB_STORAGE driver, most of the computation in the kernel space is performed by the target module, and takes more time than the IO access. Thus, the cost of adding KASAN on the module is significant (about 2X).

Besides reducing the overhead on the target module, we also verify our point of the non-intrusiveness of HART and HASAN. Running them without the target module, we find they introduce zero overhead to the main kernel, because

Table 10. Performance evaluation on KASAN with `lmbench`. HART and HASAN introduce *no* overhead to the main kernel, so the results are omitted here.

Func.	Setting	Native	KASAN	Overhead	Func.	Setting	Native	KASAN	Overhead
Processes (ms)	stat	3.08	16.4	5.3	File & VM system latency (ms)	0K File Create	44.0	136.1	3.1
	open clos	8.33	36.7	4.4		0K File Delete	35.2	227.1	6.5
	sig hndl	6.06	20.4	3.4		10K File Create	99.9	370.2	3.7
	fork proc	472	1940	4.1		10K File Delete	64.2	204.7	3.2
Local Comm. latency (ms)	Pipe	18.9	45.8	2.4		Mmap Latency	188000	385000	2.0
	AF UNIX	26.6	97.9	3.7		Prot Fault	0.5	0.5	1.0
	UDP	41.4	127.6	3.1		Page Fault	1.5	2.3	1.5
	TCP	53.4	176.4	3.3		100fd selct	6.6	13.7	2.1

they notice nothing to protect and are always sleeping. Differing from our works, `KASAN` places a significant burden on the main kernel (Table 10). This well supports the advantage of our works in terms of modification to the main kernel.

Table 11. Effectiveness evaluation on HASAN. For the result of `Detection`, "Y" means detected.

Vulnerability		Detection		
CVE-ID	Type	PoC	HASAN	KASAN
CVE-2016-0728	Use-after-free	REFCOUNT overflow [56]	Y	Y
CVE-2016-6187	Out-of-bound	Heap off-by-one [42]	Y	Y
CVE-2017-7184	Out-of-bound	xfrm_replay_verify_len [52]	Y	Y
CVE-2017-8824	Use-after-free	dccp_disconnect [33]	Y	Y
CVE-2017-2636	Double-free	n_hdlc [44]	Y	Y
CVE-2018-12929	Use-after-free	ntfs_read_locked_inode [46]	Y	Y

5.4 Effectiveness Evaluation

We present the results of our effectiveness evaluation in Table 11. As shown in this table, both HASAN and `KASAN` can detect all **6** cases. Since the selected cases cover different types of heap-related issues, including buffer overflow, off-by-one, use-after-free, and double free, we demonstrate that HASAN can cover a variety of vulnerabilities with a comparable security performance to the state-of-the-art solution in terms of the heap.

6 Discussion

Support of Other Architectures. As hardware tracing brings a significant improvement in the performance overhead during the debugging, it becomes a

common feature in major architectures. While HART achieves tracing efficiency with the support of ETM on Arm platforms, we can easily extend our design to other architectures with such feature. For example, since Processor Tracing (PT) [3], supports instruction tracing and will be enhanced with `PTWRITE` [1] to support efficient data tracing, we can extend our design by replacing the hardware set-up and configuration (given a target instrumented with `PTWRITE`).

Support of Statically Linked Modules. HART mainly targets LKM, because most of the third-party modules are released as LKM for the ease of use. Our current design cannot work with statically linked modules, mainly because we require to hook the callbacks and calls to kernel functions, which can be achieved by adjusting the relocation. However, statically linked modules are compiled and built as port of the main kernel and hence, we have no access to place the hooks. To extend support for statically linked modules, we need the kernel builder to instrument the target modules and place the required hooks.

Breakage of Perf Tools. As perf tools have already included ETM tracing capabilities, enabling tracing functionality from the user-space, HART may lead to the breakage of perf tools. We think the breakage depends on the scenario. If HART and perf are tracing/profiling different tasks, we consider perf won't be affected since HART will save the context of the hardware at the entry and exit of the traced module. However, if HART and perf are tracing/profiling the same task, perf might be affected. One easy solution is to run HART and perf side by side instead of running them concurrently.

7 Related Work

7.1 Kernel Protection

Kernel Debugger. Kmemcheck [11] can dynamically check uninitialized memory usage in x86 kernel space, but its single-stepping debugging feature drags down the speed of program execution to a large extent. KASAN [6] is the kernel version of address sanitizer. However, the heavy overhead makes it impossible to be a real-time protection mechanism, while the low overhead of HASAN show us the potential. Furthermore, KASAN requires the source code for implementation. If a vulnerability is hidden in a close-source third-party kernel module, which is common, HASAN has the potential to detect it but KASAN would not. Ftrace [10] is an software-based tracer for the kernel. Comparing with Ftrace, HART is noninvasive since we neither modifies the target kernel module nor requires the source code of the kernel module. In regard to the trace granularity, Ftrace is used to monitor function calls and kernel events with predefined tracepoints, but HART can be used to trace executed instructions and data used in these instructions. Ftrace also introduces heavy performance overhead on some event tracing, and cannot be adopted as a real-time protection as well.

Kernel Integrity Protection. KCoFI [26] enforces the CFI policy by providing a complete implementation for event handling based on Secure Virtual

Architecture (SVA) [27]. However, all software including OS should be compiled to the virtual instruction set that SVA provides. The solution also incurs high computation overhead. With slight modification to the modern architectures, Song *et al.* [53] enforces DFI over both of control and non-control data that can affect security checks. Despite the aids from the re-modelled hardware, the DFI still introduces significant latency to various system calls.

Isolation and Introspection. BGI [22] enforces isolation through an additional interposition layer between kernel and the target module. However, it requires instrumentation over source code of the target modules. Similar ideas are followed by many other works [48,58,65]. Ninja [43] is a transparent debugging and tracing framework on Arm platform, and it can also be used to introspect the OS kernel from TrustZone. However, bridging the semantic gaps between Linux kernel and TrustZone would increase the performance overhead significantly.

7.2 Hardware Feature Assisted Security Solutions

Hardware Performance Counters. HDROP [66] and SIGDROP [59] leverage PMU to count for the abnormal increase of events and seek patterns of ROP at runtime. Morpheus [38] proposes a benchmarking tool to evaluate computational signatures based mobile malware detection, in which HPCs help to create runtime traces from applications for the ease of the signature comparison. Although these solutions [37,55] achieve significant overhead reduction with the assistance of HPCs, none of them applies this hardware feature to kernel module protection.

Hardware-Based Tracing. With the availability of Program Trace Interface, Kargos [41] monitors the execution and memory access events to detect code injection attacks. However, it brings extra performance overhead due to the modification to the kernel. Since the introduction in Intel's Broadwell microarchitecture, PT has been broadly applied in solutions for software security [32,34,35,47]. All the PT-based solutions are control flow specific, differing from HART that traces both data flow and control flow for security protection.

8 Conclusion

Kernel modules demand as much security protection as the main kernel. However, the current solutions are actually limited by the requirement of source code, significant intrusiveness and heavy overhead. We present HART as a general ETM-powered tracing framework specifically for kernel modules, with the most preliminary hardware support and the most compatible method. HART can trace the selective work on the binary-only module(s) continuously by combining hardware configurations and software hooks, and decode the trace efficiently following an elastic scheduling. Based on the framework, we then build a modular security solution, HASAN, to effectively detect memory corruptions without the aforementioned limitations. Testing on a set of benchmarks in different modules, both HART and HASAN perform significantly superior to the

state-of-the-art KASAN, introducing the average overhead of **5%** and **6%**. Moreover, HASAN identifies all of the **6** vulnerabilities in different categories and modules, indicating its comparable effectiveness to the state-of-the-art solutions.

Acknowledgements. We sincerely thank our shepherd Prof. Dave Jing Tian and reviewers for their comments and feedback. This work was supported in part by grants from the Chinese National Natural Science Foundation (NSFC 61272078, NSFC 61073027).

References

1. PTWRITE - write data to a processor trace packet. https://hjlebbink.github.io/x86doc/html/PTWRITE.html
2. TCP westwood+ congestion control (2003). https://tools.ietf.org/html/rfc3649
3. Processor tracing (2013). https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing
4. Juno ARM Development Platform SoC Technical Reference Manual (2014)
5. slub (2017). https://www.kernel.org/doc/Documentation/vm/slub.txt
6. Home Google/Kasan Wiki (2018). https://github.com/google/kasan/wiki
7. Apple A4 (2019). https://www.apple.com/newsroom/2010/06/07Apple-Presents-iPhone-4/
8. Config_usb_storage: USB mass storage support (2019). https://cateee.net/lkddb/web-lkddb/USB_STORAGE.html
9. Embedded trace macrocell architecture specification (2019). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html
10. ftrace - function tracer (2019). https://www.kernel.org/doc/Documentation/trace/ftrace.txt
11. Getting started with kmemcheck - the Linux kernel documentation (2019). https://www.kernel.org/doc/html/v4.14/dev-tools/kmemcheck.html
12. H-TCP - congestion control for high delay-bandwidth product networks (2019). http://www.hamilton.ie/net/htcp.htm
13. HFS plus (2019). https://www.forensicswiki.org/wiki/HFS%2B
14. i.MX53 quick start board—NXP (2019). https://www.nxp.com/products/power-management/pmics/power-management-for-i.mx-application-processors/i.mx53-quick-start-board:IMX53QSB
15. The kernel address sanitizer (Kasan) - the Linux kernel documentation (2019). https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html
16. Kmemleak (2019). https://www.kernel.org/doc/html/v4.14/dev-tools/kmemleak.html
17. Samsung Exynos 3110 (2019). https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-3-single-3110/
18. Snapdragon 200 series (2019). https://www.qualcomm.com/snapdragon/processors/200
19. Universal disk format (2019). https://docs.oracle.com/cd/E19683-01/806-4073/fsoverview-8/index.html
20. ARM: Cortex-A8 Technical Reference Manual (2014)
21. Bigelow, D., Hobson, T., Rudd, R., Streilein, W., Okhravi, H.: Timely rerandomization for mitigating memory disclosures. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 268–279. ACM (2015)

22. Boyd-Wickizer, S., Zeldovich, N.: Tolerating malicious device drivers in Linux. In: USENIX Annual Technical Conference, Boston (2010)
23. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping kernel objects to enable systematic integrity checking. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 555–565. ACM (2009)
24. Castro, M., et al.: Fast byte-granularity software fault isolation. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pp. 45–58. ACM (2009)
25. Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., Kaashoek, M.F.: Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In: Proceedings of the Second Asia-Pacific Workshop on Systems, p. 5. ACM (2011)
26. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: complete control-flow integrity for commodity operating system kernels. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 292–307. IEEE (2014)
27. Criswell, J., Lenharth, A., Dhurjati, D., Adve, V.: Secure virtual architecture: a safe execution environment for commodity operating systems. In: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 351–366. ACM (2007)
28. Don, C., Capps, C., Sawyer, D., Lohr, J., Dowding, G., et al.: IOzone filesystem benchmark (2016). http://www.iozone.org/
29. Dugan, J., Elliott, S., Mah, B.A., Poskanzer, J., Prabhu, K., et al.: iPerf - the ultimate speed test tool for TCP, UDP and SCTP (2018). https://iperf.fr/
30. Floyd, S.: Highspeed TCP for large congestion windows (2003). https://tools.ietf.org/html/rfc3649
31. Garfinkel, T., Rosenblum, M., et al.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, pp. 191–206 (2003)
32. Ge, X., Cui, W., Jaeger, T.: GRIFFIN: guarding control flows using intel processor trace. In: Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2017)
33. Ghannam, M.: CVE-2017-8824 Linux: use-after-free in DCCP code (2017). https://www.openwall.com/lists/oss-security/2017/12/05/1
34. Gu, Y., Zhao, Q., Zhang, Y., Lin, Z.: PT-CFI: transparent backward-edge control flow violation detection using intel processor trace. In: Proceedings of the 7th ACM International Conference on Data and Application Security and Privacy (CODASPY) (2017)
35. Hertz, J., Newsham, T.: Project triforce: run AFL on everything! (2016). https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/
36. Hinum, K.: Hisilicon kirin 920 (2017). https://www.notebookcheck.net/HiSilicon-Kirin-920-SoC-Benchmarks-and-Specs.240088.0.html
37. Iyer, R.K.: An OS-level framework for providing application-aware reliability. In: 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006 (2007)
38. Kazdagli, M., Ling, H., Reddi, V., Tiwari, M.: Morpheus: benchmarking computational diversity in mobile malware. In: Proceedings of Hardware and Architectural Support for Security and Privacy (2014)
39. Linares-Vásquez, M., Bavota, G., Escobar-Velásquez, C.: An empirical study on android-related vulnerabilities. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 2–13. IEEE (2017)

40. Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., Vigna, G.: Dr. Checker: a soundy analysis for Linux kernel drivers. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 1007–1024. USENIX Association (2017)
41. Moon, H., Lee, J., Hwang, D., Jung, S., Seo, J., Paek, Y.: Architectural supports to protect OS kernels from code-injection attacks. In: Proceedings of Hardware and Architectural Support for Security and Privacy (2016)
42. Nikolenko, V.: Heap off-by-one POC (2016). http://cyseclabs.com/exploits/matreshka.c
43. Ning, Z., Zhang, F.: Ninja: towards transparent tracing and debugging on arm. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 33–49 (2017)
44. Popov, A.: CVE-2017-2636: exploit the race condition in the n_hdlc Linux kernel driver bypassing SMEP (2017). https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html
45. Rubin, P., MacKenzie, D., Kemp, S.: dd - convert and copy a file (2019). http://man7.org/linux/man-pages/man1/dd.1.html
46. Schumilo, S.: Multiple memory corruption issues in ntfs.ko (Linux 4.15.0-15.16) (2018). https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1763403
47. Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T.: kAFL: hardware-assisted feedback fuzzing for OS kernels. In: Proceedings of the 26th Security Symposium (USENIX Security) (2017)
48. Sehr, D., et al.: Adapting software fault isolation to contemporary CPU architectures. In: 19th USENIX Security Symposium (USENIX Security 2010), pp. 1–12 (2010)
49. Freescale Semiconductor: i.MX53 Multimedia Applications Processor Reference Manual (2012)
50. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: a fast address sanity checker. In: USENIX Annual Technical Conference, pp. 309–318 (2012)
51. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: ACM SIGOPS Operating Systems Review, pp. 335–350. ACM (2007)
52. snorez: Exploit of CVE-2017-7184 (2017). https://raw.githubusercontent.com/snorez/exploits/master/cve-2017-7184/exp.c
53. Song, C., Lee, B., Lu, K., Harris, W., Kim, T., Lee, W.: Enforcing kernel security invariants with data flow integrity. In: NDSS (2016)
54. Swift, M.M., Martin, S., Levy, H.M., Eggers, S.J.: Nooks: an architecture for reliable device drivers. In: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, pp. 102–107. ACM (2002)
55. Tang, A., Sethumadhavan, S., Stolfo, S.J.: Unsupervised anomaly-based malware detection using hardware features. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) RAID 2014. LNCS, vol. 8688, pp. 109–129. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11379-1_6
56. Perception Point Team: Refcount overflow exploit (2017). https://github.com/SecWiki/linux-kernel-exploits/blob/master/2016/CVE-2016-0728/cve-2016-0728.c
57. virtuoso: virtuoso/etm2human: Arm's ETM v3 decoder (2009). https://github.com/virtuoso/etm2human
58. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: ACM SIGOPS Operating Systems Review, pp. 203–216. ACM (1994)
59. Wang, X., Backer, J.: SIGDROP: signature-based ROP detection using hardware performance counters. arXiv preprint arXiv:1609.02667 (2016)

60. Wang, Z., Jiang, X.: HyperSafe: a lightweight approach to provide lifetime hypervisor control-flow integrity. In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 380–395. IEEE (2010)
61. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 545–554. ACM (2009)
62. Wu, W., Chen, Y., Xing, X., Zou, W.: Kepler: facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities. In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 1187–1204 (2019)
63. Wu, W., Chen, Y., Xu, J., Xing, X., Gong, X., Zou, W.: Fuze: towards facilitating exploit generation for kernel use-after-free vulnerabilities. In: 27th USENIX Security Symposium (USENIX Security 2018), pp. 781–797 (2018)
64. Xiong, X., Tian, D., Liu, P., et al.: Practical protection of kernel integrity for commodity OS from untrusted extensions. In: NDSS, vol. 11 (2011)
65. Zhou, F., et al.: SafeDrive: safe and recoverable extensions using language-based techniques. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 45–60. USENIX Association (2006)
66. Zhou, H.W., Wu, X., Shi, W.C., Yuan, J.H., Liang, B.: HDROP: detecting ROP attacks using performance monitoring counters. In: Huang, X., Zhou, J. (eds.) ISPEC 2014. LNCS, vol. 8434, pp. 172–186. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06320-1_14