DriverJar: Lightweight Device Driver Isolation for ARM

Abstract—Driver-originated vulnerabilities are well-known threats to modern monolithic kernels. However, existing driver isolation solutions either rely on Intel-only or newly-introduced CPU features (e.g., Intel VMFUNC, ARM MTE), or suffer from performance issues, making them unsuitable for existing ARM-based devices. In this work, we leverage a common hardware feature, named hardware watchpoint, to achieve lightweight driver isolation for off-the-shelf ARM devices. Specifically, we utilize watchpoints to prevent the possibly compromised driver from corrupting the rest kernel’s state arbitrarily. We implement a prototype for ARM64 Linux. The security analysis and performance evaluation show the efficiency and practicality of our solution.

Index Terms—Driver Isolation, ARM, Watchpoint, Software Fault Isolation

I. INTRODUCTION

Nowadays, a large number of device drivers have been developed to extend the functionality and hardware support of the OS kernel. However, since the quality and stability of device drivers are often inferior to the core kernel, they are more likely to contain vulnerabilities compared to the core kernel. Previous study [1] indicates that about 2/3 of Linux kernel vulnerabilities originates from kernel modules or device drivers. Nevertheless, since modern operating system kernel is monolithic, a driver-originated vulnerability could be exploited to attack the entire kernel.

To address the security threats posed by device drivers, one prevailing approach is driver isolation, which falls into the following categories. First, researchers have proposed a series of virtualization and hypervisor-based solutions to achieve secure separation of device drivers and kernels. However, these solutions introduce considerable context-switching overhead, which can significantly impact the performance of low-powered devices. Second, to this end, some studies have implemented low-overhead isolation mechanisms based on isolation primitives provided by the CPU. On the Intel platform, researchers have proposed driver isolation solutions based on Extended Page Table(EPT), VM Functions(VMFUNC), and Memory Protection Keys(MPK). As for the ARM platform, existing hardware-assisted driver isolation solutions rely on features that are either only supported by the latest processors(i.e., Pointer Authentication, Memory Tagging) or deprecated(i.e., Domain Access Control), preventing them from being adopted on most existing devices.

In this paper, we propose DriverJar, a hardware-assisted isolation framework for protecting the kernel from driver-originated memory corruption and exploitations. The basic idea is to divide the (untrusted) device driver and trusted base kernel into separate domains and leverage hardware watchpoint, a commonly supported debugging feature, to restrict the data access from the isolated driver to the base kernel. First, to enforce the protection, each cross-domain function call between the isolated driver and rest kernel must invoke a trampoline, which contains secure gates for updating the protection state. Second, to avoid the protection being disabled or bypassed, we carefully design the secure gates and ensure that the adversary cannot compromise the protection using interrupts, exceptions, or privileged instructions. Third, we provide a data-write trampoline containing an allowlist check for legitimate kernel object updates from the isolated driver. In addition, our solution takes into account the isolated driver’s need for dynamic memory allocation.

We have developed a prototype called DriverJar for AArch64 Linux 5.4.117 and evaluated it on a Hikey970 development board [2]. We first performed a security analysis to show the security guarantees of our system cannot be bypassed. To measure the performance overhead caused by DriverJar, we conduct a series of experiments. First, we measure the cycle count required for domain switches. We then benchmarked the modified kernel, and the results show that the changes we made to the kernel caused small performance degradation. Finally, our benchmark of the isolated dummy driver shows that DriverJar brings a performance degradation of 14%-17% (the high overhead bound), which is comparable to some state-of-the-art solutions.

II. BACKGROUND AND RELATED WORK
A. Device Driver Isolation

Over the years, researchers have explored various ways to separate device drivers from the monolithic kernel, including isolating drivers to userspace [3] [4], approaches that invoke virtualization or hypervisor [5]–[7], approaches based on Software Fault Isolation(SFI) [8]–[10], and more. However, these solutions introduce prohibitive context-switching overheads. According to [11], a hypervisor call(HVC) or secure monitor call(SMC) consumes hundreds of cycle. Meanwhile, traditional SFI solutions such as Nooks [8] can cause a 72% performance degradation. In order to improve the performance of driver isolation, several isolation solutions that take advantage of processor hardware features have been proposed. For instance, LVDs [12] proposed a lightweight driver isolation technique based on Intel EPT and VMFUNC. The addition of Pointer Authentication(PAC) [13] and Memory Tagging Extension(MTE) [14] extension on the latest ARM processors also makes hardware-assisted SFI solutions possible [15].
Moreover, a driver isolation solution based on Domain Access Control (DAC) has also been proposed [16]. However, since PAC and MTE are currently only supported by the newest ARM processors, and DAC is a 32-bit only feature, none of these solutions can be applied to most existing ARM devices.

B. ARM Watchpoint

Hardware watchpoint is a common self-hosted debugging mechanisms used to monitor data access to specific memory regions. Depending on the configuration, a specific type of access to the monitored memory region incurs a watchpoint exception, which will be caught and handled by privileged software, such as the OS kernel. The watchpoint feature on ARM has been used to implement a variety of security applications. Jang et al. proposed an in-process memory isolation solution based on watchpoint [17]. They further proposed watchpoint-based emulation of privileged access never (PAN) and kernel execute-only memory (XOM) [18]. In addition, SelMon [19] uses watchpoint and data execution prevention (DEP) to help implement a self-protected kernel integrity monitor.

For current ARM processors, the maximum number of watchpoint-monitored regions is SoC-dependent and can be up to 16. Each watchpoint-monitored region is configured by setting its corresponding debug watchpoint value register (DBGWVR(n)_EL1, n = 0-15) and debug watchpoint control register (DBGWCR(n)_EL1, n = 0-15). These watchpoint-related registers are per-core registers, which enables developers to perform debugging on thread basis. The watchpoint value register (DBGWVR) sets up the starting address of the monitored region. The watchpoint control register (DBGWCR) comprises important attributes of the monitored region. Specifically, the BAS and MASK flags determine the monitoring granular and size respectively; The combination of the security state control (SSC) and the privilege of access control (PAC) flags is used to define the security state and exception level under which the exception should be generated; The load store control (LSC) flag determines the type of the monitored access type as read (0b01), write (0b10), or both (0b11). In addition to the settings for the watchpoint registers, the monitor debug events (MDE) flag of the monitor debug system control register (MDSCR_EL1) has to be set to activate the monitoring.

It should be noted that the watchpoint configuration must strictly comply with the monitoring size and address-alignment requirements. Due to the way the MASK flag is set, the size of each monitored region is a power of 2. If the size is less or equal to 8 bytes, the starting address of monitoring must be aligned with a word or double word; Otherwise, the starting address of the monitored region must be aligned with the monitoring size. If this requirement is not satisfied, the watchpoint will not generate any exception.

III. Assumptions and Threat Model

In our threat model, we assume the device driver may contain flaws (such as memory corruption vulnerabilities) that could be compromised by an adversary. Our goal is to protect the state of the rest kernel from the adversary’s further corruption by isolating the vulnerable driver in a separate domain. In addition, we assume Privilege Access Never (PAN) and Privilege eXecute Never (PXN) are enabled in the kernel so that the compromised driver cannot access userspace content or leverage userspace for further corruption. Besides, a secure boot is also assumed to ensure the benign initial state of the kernel. Finally, we consider both side-channel and speculative-execution attacks as out of scope. Defending against such attacks is orthogonal to our work.

IV. System Design

A. Overview

![Diagram of driver isolation mechanism](image)

DriverJar aims to prevent an adversary who has compromised one vulnerable device driver from corrupting the rest kernel’s data integrity by hijacking control flow or overwriting sensitive kernel objects. To this end, DriverJar utilizes a common hardware feature, namely the hardware watchpoint, to enforce the isolation between the possibly flawed driver and the rest kernel. Specifically, we put the isolated driver and the rest kernel into separated domains. When the execution gets into the isolated driver, DriverJar will set up watchpoints to monitor the data access to the rest kernel’s memory region. Any data written to the rest kernel must be authorized and proxied. Direct or illegal data written from the isolated driver would incur a watchpoint exception and thus be caught by the corresponding handler. Watchpoint monitoring is only disabled when the control flow leaves the isolated driver. In this way, even though the isolated driver is compromised, the rest of the kernel could still keep safe from data corruption.

However, it is non-trivial to enforce such isolation. Two aspects of security designs need to be considered to ensure that our watchpoint monitoring cannot be bypassed.

- **Control Flow Security** We enable watchpoint monitoring before the control flow transfers to the isolated driver. As the monitoring is enabled, it cannot be disabled or bypassed by the isolated device driver.

- **Data Flow Security** Since the isolated driver may require updating data state that does not belong to itself, any data written from the isolated driver to the rest kernel has to
Listing 1: secure gates for cross-domain calls

```c
entry_gate:
    /* Entry gate begins */
    disable_irq
    switch_to_driver_stack
    enable_watchpoint_monitoring
    restore_irq
    /* End of entry gate */

exit_gate:
    /* Exit gate */
    disable_irq
    switch_to_kernel_stack
    exit_gate:
```

be authorized, and the least-privilege principle should be enforced.

In the following, we will illustrate our control flow and data flow security designs, respectively. Fig. 1 shows the overall system architecture of DriverJar.

B. Control Flow Security

To alter the protection state when the domain transition occurs, secure gates (including entry/exit gate) are set up to handle cross-domain calls between the isolated driver and the rest kernel. The entry gate is responsible for the kernel-to-driver domain switch (including watchpoint setup) while the exit gate does the opposite.

As shown in Listing 1, we have carefully designed our secure gates to avoid being exploited by attackers. The core idea is to ensure the execution after disabling the watchpoint monitoring in the exit gate cannot be manipulated by the possibly compromised driver.

First, instead of reusing the kernel stack, DriverJar allocates a separate driver stack on demand for the driver execution. The kernel/driver stack switch is done in the secure gates. Note that the original stack pointer will be saved into the task struct for restoration later to support nested cross-domain calls. Using a separate driver stack brings two benefits. On the one hand, the memory used by the kernel stack is monitored by the watchpoint during the driver execution (cannot be corrupted by the driver execution), so it is necessary to switch to a writable stack for functionality. On the other hand, this blocks the possibly compromised driver from interfering with the execution after leaving the isolated driver by corrupting stack states.

Second, in the code enable_wp_monitoring and disable_wp_monitoring, enabling and disabling watchpoint monitoring operations are followed by additional instructions (i.e., cmp and b.ne) to double-check the watchpoints are set properly. Such a design blocks the exploitation of watchpoint-related system register update instructions in the secure gates by a control-flow hijacking attack, i.e., directly jumping to the instruction in the secure gates to disable the watchpoint. We borrow this design trick from widely discussed MPK-based security hardening solutions (XXXX references).

Third, we include memory barriers and interrupt control instructions in secure gates to prevent malicious interrupts and side effects of out-of-order execution. Last but not least, each operation in the secure gates is implemented as an inline assembly function, and there is no indirect control flow transfer in the secure gates to present control hijacking.

Apart from the explicit cross-domain function calls, interrupts and exceptions can also cause the control flow transfers between the two domains. Therefore, we modified the entry.s file, which contains the entry/exit for interrupt/exception handling. When an interrupt or exception occurs during the driver execution, DriverJar will save the current watchpoint state and the stack pointer, disable watchpoint monitoring and switch to the kernel stack before handling the interrupt/exception. When the interrupt/exception handler returns, DriverJar restores the original watchpoint state and stack pointer if needed. Two benefits are brought by such a design. First, it allows driver function invocation (which requires domain switching) in the interrupt handling, thus satisfying the kernel’s functionality requirement. Second, it makes DriverJar transparent to the kernel scheduler and thus no intrusive modification for the kernel scheduler.

Furthermore, DriverJar ensures that there are no non-secure watchpoint-related system register updates inside the kernel that could be exploited by the compromised driver. To this end, DriverJar performs a code inspection for the driver at the load time to ensure there are no watchpoint-related system register update instructions other than secure gates. As far as we know, watchpoint is seldom used by drivers in the production environment (it’s a debug feature), disallow the existence of the watchpoint-related instructions in the driver will not impede the driver’s functionality. Moreover, we patch the core kernel (mainly hw_breakpoint.c) to ensure the occurrence of the corresponding instructions is secure.

By combining all the above designs, DriverJar ensures that watchpoint monitoring could not be disabled or bypassed by the possibly compromised driver and thus control flow security is enforced.

C. Data Flow Security

The isolated driver may need to update kernel objects during its execution. To support this, DriverJar introduces the data write trampoline, which wraps normal data write with additional operations (such as watchpoint monitoring switch). Moreover, to block the possibly compromised driver from abusing the driver’s data write trampolines, we include a dynamic allowlist check in our design. Allowlist is used in the data write trampoline to check whether the target kernel object is allowed access. During driver development, there would only a well-defined subset of kernel objects that are necessary for the driver to function would be included in the allowlist. Furthermore, the allowlist is designed to get updated on demand during its lifecycle so that the least privilege principle is always enforced. Specifically, before the kernel calls an isolated driver function for the first time, an allowlist is initialized by DriverJar. Whenever the isolated driver invokes the kernel function API during its execution, DriverJar will update the allowlist on demand if needed. The
#define NEW_VALUE(dst, op, ...) ({
    typeof(dst) _dst = (dst); \
    _dst op __VA_ARGS__; })

#define SECURE_WRITE(dst, len, op, ...) do {
    netdev_tx_t _ret = dummy_xmit(arg0, arg1);
    _dst op __VA_ARGS__; }

static void
allowlist_init();

static void
allowlist_free();

Listing 2: Design and usage of the data-write trampoline.

### TABLE I
**ANNOTATION MACROS PROVIDED FOR DRIVER DEVELOPERS**

<table>
<thead>
<tr>
<th>Annotation macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRAP_n</td>
<td>Generate wrapper for driver function</td>
</tr>
<tr>
<td>IMPORT_n</td>
<td>Generate wrapper for kernel function</td>
</tr>
<tr>
<td>SECURE_WRITE()</td>
<td>Perform a data write with allowlist check</td>
</tr>
<tr>
<td>ALLOW()</td>
<td>Append an allowlist entry</td>
</tr>
<tr>
<td>CHECK()</td>
<td>Check whether a function parameter is legal</td>
</tr>
<tr>
<td>PURGE()</td>
<td>Remove an allowlist entry</td>
</tr>
</tbody>
</table>

1 n means there are n parameters for the function.

allowlist would be released after the initial driver function invocation is finished. In addition, the allowlist is implemented as a hash table for constant-time lookup and stored in the `task_struct` since it couples with the thread execution.

Listing 2 shows the design and usage of our data-write trampoline. To perform a kernel object update, the trampoline first calculates the new value of the kernel object. Then, the trampoline temporarily disables watchpoint monitoring and checks whether the data write is legal with the allowlist (i.e. `is_permitted`). If the starting address and length match an allowlist entry, the trampoline applies the new value and re-enables watchpoint monitoring.

D. Dynamic memory allocation for driver

Apart from kernel data integrity, DriverJar also needs to ensure that the device driver can still function properly after being isolated. Since we do not restrict data reads and kernel function invocation is supported, most of the external interaction needs of the isolated device driver can be met. However, we also need to consider the operational requirements of the device driver’s code, such as dynamic memory allocation. To solve this challenge, DriverJar chooses to maintain a dedicated memory pool for each isolated driver. The memory pool is located in the driver’s memory region for direct access and common memory management primitives (e.g., `kmalloc`, `vmalloc`) are provided for easy-to-use.

E. Developer tools

Table I shows the annotation macros we provided to allow driver developers to adapt DriverJar for their driver. Specifically, `WRAP_n`, `IMPORT_n`, `ALLOW`, `CHECK`, `PURGE` are used for the generation of driver/kernel function wrappers, which wrap the original driver/kernel function to perform cross-domain calls. There are mainly two tasks for the wrappers: (1) watchpoint monitoring enable/disable by leveraging secure gates. (2) allowlist management including initialization, append, and removal. In addition, function parameters could also be checked within the wrapper with the annotation of `CHECK`. Listing 3 shows an example usage of our annotation macros for the kernel/driver function APIs and the corresponding generated wrappers. As for `SECURE_WRITE` annotation macro, it generates data-write trampolines for kernel object updates. An example usage could be found in Listing 2.

At the development stage, driver developers could leverage the above-provided annotation macros to define the driver/kernel interaction behaviors. Then, the corresponding driver/kernel function wrappers and data-write trampolines would be generated based on the developer’s annotation and thus DriverJar’s isolation is enabled.

V. IMPLEMENTATION DETAILS

We have implemented a prototype of DriverJar on a Hikey970 development board based on AArch64 Linux kernel version 5.4.117. Currently, our prototype only supports one isolation domain for the kernel drivers. It is because of the limited number of watchpoint counts (four pairs) in
the development board. Our design could be extended to provide more isolation domains for kernel drivers as long as more watchpoints are provided. Detailed discussion about this is illustrated in ?? In this section, we illustrate some implementation details not covered before.

We adjust the kernel memory layout to meet the watchpoint’s alignment requirements and achieve complete protection of kernel data regions. The adjusted kernel space memory layout is shown in Fig. 2. First, we reduce the size of the vmalloc area to 4GB and two watchpoints are used to monitor it during driver isolation. To get vmalloc area start address 2GB-aligned, we expand the size of modules area from 128MB to 1920MB. The remaining vmalloc area is used as the isolated driver’s data region. Then, we use a 2GB watchpoint to monitor fixed, PCI I/O and vmemmaps area (shown as the “misc.” area in Fig. 2). Finally, we shrink the direct mapping region of the kernel (i.e., memory area in Fig. 2) to 2GB so that it could be fully covered by the remaining watchpoint. Note that we make the above restrictions due to the lack of watchpoints in our development board. In another word, these restrictions could be released with more watchpoints provided.

In addition, we implement our dedicated memory pool for the isolated driver based on Linux genpool mechanism. We allocate the memory pool from the driver’s memory region. Moreover, the different granularity of memory allocation is supported for the memory pool to avoid memory fragmentation problems and improve performance.

VI. Security Analysis

The security of DriverJar mainly depends on the enforcement of watchpoint monitoring during driver execution. Since the core kernel is patched to ensure the occurrence of watchpoint update instructions is all sanitized and DriverJar verifies there is no watchpoint update instruction in the driver at load time, the possibly compromised driver could only disable watchpoint monitoring by leveraging the secure gates. As described before, DriverJar has carefully designed the secure gates to ensure the execution cannot be manipulated by the possibly compromised driver after the secure gates disable watchpoint monitoring. Thus, the secure gates could not be manipulated to bypass watchpoint monitoring during driver execution.

The possibly compromised driver may corrupt kernel data integrity by manipulating kernel objects specified in the allowlist, since allowlist configuration highly relies on the developer’s domain knowledge. To mitigate this threat, we provide a static analysis tool based on libclang [20] to help developers with their allowlist configuration. We regard exploring automatic allowlist generation as our future.

VII. Performance Evaluation

In this section, we measure the performance overhead of DriverJar on a development board. The experiments have been conducted on the 96boards Hikey970 platform [2], which ships with a Cortex-A73 2.36GHz quad-core processor and Cortex-A53 1.8GHz quad-core processor in a big.LITTLE design and 6GB of DRAM. Considering that DriverJar may have different performance effects on big and small cores, we conduct the following experiments on big and small cores separately.

A. Switching Overhead

To investigate the performance overhead imposed by the domain switching, we use the ARM performance counter to get the required cycle count of each cross-domain function call and the data written. In each operation, we count the cycle count of 100 empty function calls and the difference between the cycle count of 100 normal data writes and 100 wrapped data writes. Moreover, we repeat each operation 10 times for accurate results. The average results are reported in Table II, which shows the efficiency of our trampoline implementation.

B. OS Micro Benchmark

Apart from domain switching, DriverJar also imposes a performance penalty on basic OS operations, as we made code changes on interrupt handling and task creation. We measured such overhead using the LMBench test suite. Table III reports the results of the experiments, which indicates that the changes we made to the kernel barely impacts the OS performance except for some operations related to task creation. This performance degradation is expected as the kernel allocates driver stack for each newly created task_struct, regardless of whether the corresponding task will execute any code from the isolated driver.
C. Isolated Device Driver Benchmark

We use the dummy driver to measure the performance overhead when our solution is applied to a "fast" device driver. Dummy is a software-only driver that emulates an infinitely fast network adapter, which allows us to stress the performance overhead without hitting any artificial hardware limits. In this experiment, we use three versions of the dummy driver, one unmodified and the other two isolated, with allowlist checking enabled and disabled respectively. By comparing the performance test results of the isolated drivers with those of the original dummy driver, we can see how much the domain switching and allowlist checking affect the performance. We use the iperf2 benchmark to measure the transmit bandwidth of the MTU-sized packets.

We report total packet transmission I/O requests per second (IOPS) across all CPU cores, as depicted in Fig. 3. When using only one big core for testing, the non-isolated driver achieves 281K IOPS, and the isolated drivers achieve 248K IOPS (88.4% of the non-isolated performance) and 234K IOPS (83.6% of the non-isolated performance), depending on whether the allowlist checking is enabled. When testing on a single little core, the non-isolated driver achieves 118K IOPS, and the isolated drivers achieve 110K IOPS (93.5% of the non-isolated performance) and 100K IOPS (85.5% of the non-isolated performance) respectively. With all cores utilized, the non-isolated can achieve 1340K IOPS, and the isolated drivers achieve 1238K (92.4% of the non-isolated performance) and 1151K IOPS (89.5% of the non-isolated performance) respectively. In sum, DriverJar delivers a 14%-17% performance reduction for the isolated dummy driver. Considering that the dummy driver does not include any data processing operation in the packet-sending process, we expect a lower performance reduction when DriverJar is applied to real-world device drivers.

In addition, the results also show that the efficiency of our solution is comparable to state-of-the-art hardware-assisted solutions. For instance, HAKC, an isolation solution based on PAC and MTE, estimated a 20% maximum performance degradation in its experiments with ipv6.ko. The LVDs solution, which is based on EPT and VMFUNC, also used a software-only network device driver to measure its high-bound performance overhead. In single-threaded tests, LVDs introduce a performance reduction of 28%-35%, depending on whether the processor’s extended state needs to be saved or not.

VIII. DISCUSSION

Limited number of watchpoints. Due to the limited availability of watchpoints, our solution can only provide complete protection for devices with smaller memory capacity. However, we also realize that a lot of critical kernel data does not change after device initialization is complete. Therefore, by profiling kernel memory usage, our solution can still be used to protect some of the most critical kernel data, despite the insufficient number of watchpoints. In addition, we believe our solution will be instructive for some hardware-assisted memory protection features that may emerge in the future.

Porting efforts. During the driver porting progress, we realize that the process of generating and replacing wrappers for kernel functions called by device drivers can be largely automated. To facilitate pointer replacement and wrapping cross-domain data writes, we implemented a source code analysis tool based on libclang [20] for identifying control flow graphs. Our tool is effective, but manual efforts cannot be completely avoided. In general, modifying existing device drivers to apply our solution is still labor-intensive and requires some understanding of how that device driver works.

Optimizations. For kernel functions that do not use the stack, such as atomic operations, we remove the switch stack...
operation from their wrapper to reduce performance loss. In addition, we allocate isolated stacks for each task_struct during its initialization to avoid stack allocation failures since vmalloc functions are not usable in interrupts. The overhead caused by allocating the driver stack can be further reduced by preparing a memory pool dedicated to driver stack allocation and only allocating the stack for tasks that execute driver code.

IX. CONCLUSION

We present DriverJar, an hardware-assisted driver isolation solution for ARM devices. DriverJar utilizes hardware watchpoint to restrict isolated driver’s data access to the kernel. By mediating cross-domain control and data flows, we ensure that the isolated device drivers cannot disable or bypass the protection while working properly. In addition, we ensure the isolated driver works properly. We describe how to integrate our solution into the driver using developer tools. The performance evaluation show that our solution introduces a small loss in performance.

REFERENCES