# DriverJar: Lightweight Device Driver Isolation for ARM

Huamao Wu*, Yuan Chen*, Yajin Zhou†
Zhejiang University, Hangzhou, China

Yifei Wang, Lubo Zhang
Huawei Technologies Co. ltd, Beijing, China

*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 originate from kernel modules or device drivers. Nevertheless, since most modern operating system kernels are monolithic, a driver-originated vulnerability could be exploited to attack the entire kernel.

One prevailing approach to address the security threats posed by device drivers is driver isolation, which falls into the following categories. First, researchers have proposed a series of software-based solutions [2]–[6] to achieve secure separation between device drivers and kernels. However, these solutions introduce considerable context-switching overhead, which can significantly impact the performance of low-powered devices. Second, some studies have implemented low-overhead isolation mechanisms based on isolation primitives provided by the CPU. On the Intel platform, researchers have proposed isolation solutions based on Extended Page Table (EPT) [7] and VM Functions (VMFUNC) [8]. As for the ARM platform, existing hardware-assisted driver isolation solutions [9] [10] 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 used 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) core kernel into separate domains and leverage hardware watchpoint, a commonly supported debugging feature, to restrict the data access from the isolated driver to the rest kernel. First, to enforce the protection, each cross-domain function call between the isolated driver and rest kernel must invoke a trampoline containing secure gates for watchpoint monitoring updates. Second, to avoid the watchpoint monitoring 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 [11]. To evaluate DriverJar, we first performed a security analysis to show that an adversary cannot bypass the security guarantees of our system. Then, we conducted experiments to measure the performance overhead caused by DriverJar. First, we measured 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 minor performance degradation. Finally, we performed a stress test on `dummy`, a software-only network driver. The results show that DriverJar brings a performance degradation of 14%-17%, 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 [5] [6], invoking virtualization [12]–[15], adopting Software Fault Isolation (SFI) techniques [2]–[4], and more. However, these solutions introduce prohibitive context-switching overhead. For example, instrumenting a network driver with LXFI [4], an SFI-based solution, increases CPU usage by 2.2–3.7× and introduces a 35% performance loss while transmitting UDP packets. 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 [8] proposed a lightweight driver isolation technique based on Intel EPT and VMFUNC. The addition of Pointer Authentication (PAC) [16] and Memory Tagging Extension (MTE) [17] extension on the latest ARM processors also makes hardware-assisted SFI solutions possible [10]. Moreover, a driver isolation solution based on Domain Access Control (DAC) [9] has also been proposed. However, since PAC and MTE are currently supported by the newest ARM processors only, 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 mechanism 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 various security applications. Jang et al. proposed an in-process memory isolation solution based on watchpoint [18]. They further offered watchpoint-based emulation of privileged access never (PAN) and kernel execute-only memory (XOM) [19]. In addition, SelMon [20] uses watchpoint and data execution prevention (DEP) to help implement a self-protected kernel integrity monitor.

---

†: Corresponding author(yajin_zhou@zju.edu.cn).
*: These authors contributed equally to the work.

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 enable developers to perform debugging on a 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 granularity 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 eXecute Never (PXN) are enabled in the kernel so that the compromised driver cannot 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

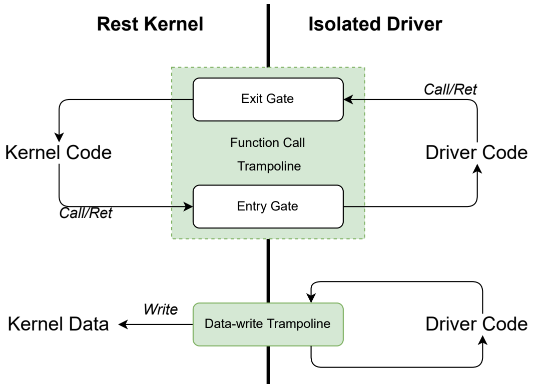

Fig. 1. The system architecture of DriverJar.

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 update to the rest kernel must be authorized and proxied. Direct or illegal data updates 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, the rest kernel could keep safe from data corruption even if the isolated driver is compromised.

```
entry_gate:
    disable_irq
    switch_to_driver_stack
    enable_wp_monitoring
    restore_irq

exit_gate:
    disable_irq
    disable_wp_monitoring
    switch_to_kernel_stack
    restore_irq
```

Listing 1: Secure gates for cross-domain function calls.

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

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

### B. Control Flow Security

To alter the protection state when the domain transition occurs, we set up secure gates (including entry/exit gates) to handle cross-domain calls between the isolated driver and the rest kernel. The entry gates are responsible for the kernel-to-driver domain switch (including watchpoint setup), while the exit gates 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 secure gates 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 the widely discussed MPK-based security hardening solutions [21], [22].

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 prevent control hijacking.

Apart from the explicit cross-domain function calls, interrupts and exceptions also need to be considered for domain switching. Because handling interrupt or exception with watchpoint monitoring and driver stack could bring functionality issues and possibly crash the kernel. 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. In this way, interrupt/exception handling functions correctly as before. Moreover, 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 the compromised driver could exploit. To this end, DriverJar performs a code inspection for the driver at the load time to ensure 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 as it is a debugging feature. Therefore, disallowing the existence of 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 cannot 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. Therefore, DriverJar introduces the data write trampoline, which wraps normal data updates 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, only a well-defined subset of kernel objects 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 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. The pointer of the allowlist is stored in the `task_struct` since it couples with the thread execution.

```
#define NEW_VALUE(dst, op, ...) ({\
    typeof(dst) _dst = (dst); \
    _dst op __VA_ARGS__; })

#define SECURE_WRITE(dst, len, op, ...) do {\
    typeof(dst) _dst = NEW_VALUE((dst), op, __VA_ARGS__); \
    disable_wp();   \
    if (is_permitted(&(dst), (len))) (dst) = _dst;    \
    enable_wp(); } while (0)

// Example: updating dest (int from kernel).
// *dest = source + 4;
SECURE_WRITE(*dest, sizeof(int), =, source + 4);
// *dest ++;
SECURE_WRITE(*dest, sizeof(int), ++);
```

Listing 2: Design and usage of the *SECURE_WRITE* macro, which generates a data write trampoline.

TABLE I
ANNOTATION MACROS PROVIDED FOR DRIVER DEVELOPERS

| Annotation macro | Description |
|---|---|
| WRAP_n()[1] | Generate wrapper for driver function |
| IMPORT_n()[1] | Generate wrapper for kernel function |
| SECURE_WRITE() | Perform a data update with allowlist check |
| ALLOW() | Append an allowlist entry |
| CHECK() | Check whether a function parameter is legal |
| PURGE() | Remove an allowlist entry |

[1] n means there are *n* parameters for the function.

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 update 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. In addition, 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) enable/disable the watchpoint monitoring by leveraging secure gates. (2) allowlist management, including initialization, append, and removal. In addition, function parameters

```
/*
static netdev_tx_t dummy_xmit(struct sk_buff *skb,
                             struct net_device *dev)
*/
static WRAP_2(dummy_xmit, netdev_tx_t,
  struct sk_buff *, struct net_device *,
// Allowlist operations before call(if any)
  ALLOW(arg0, sizeof(struct sk_buff))
  ALLOW(arg1, sizeof(struct net_device)),
// Allowlist operations after call(if any)
  DO_NOTHING()
)
/* Generated wrapper for dummy_xmit() */
static netdev_tx_t dummy_xmit_wrapper(
  struct sk_buff *arg0, struct net_device *arg1)
{
    allowlist_init();
    ALLOW(arg0, sizeof(struct sk_buff));
    ALLOW(arg1, sizeof(struct net_device));
    entry_gate();
    netdev_tx_t _ret = dummy_xmit(arg0, arg1);
    exit_gate();
    allowlist_free();
    return _ret;
}
/* void kfree(const void *x) */
static IMPORT_VOID_1(kfree,
  const void *,
// Allowlist operations before call(if any)
  CHECK(arg0),
// Allowlist operations after call(if any)
  PURGE(arg0)
)
/* Generated wrapper for kfree */
static void kfree_wrapper(const void *arg0)
{
    exit_gate();
    CHECK(arg0);
    kfree(arg0);
    PURGE(arg0);
    entry_gate();
    return;
}
```

Listing 3: Annotation macro usage examples.

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 Section VIII. 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.

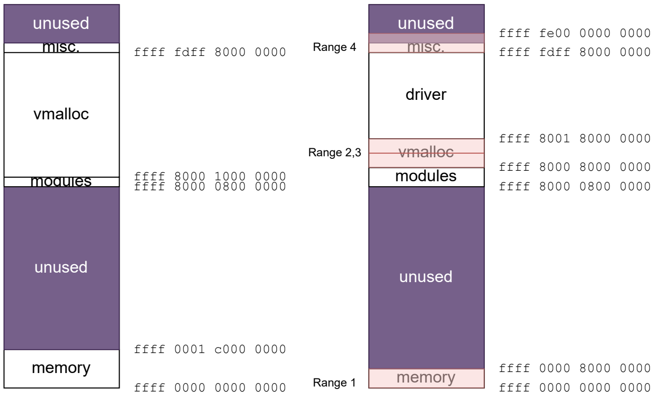

Fig. 2. The original and modified kernel memory layout of the Hikey970 board. The watchpoint-protected areas are marked in red.

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 possibly compromised driver cannot manipulate the execution 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 [23] to help developers with their allowlist configuration. We regard exploring automatic allowlist generation as our future work.

## VII. PERFORMANCE EVAULATION

In this section, we measure the performance overhead of DriverJar on a development board. The experiments have been conducted on the 96boards Hikey970 platform [11], which ships with a Cortex-A73 2.36GHz quad-core processor and a 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

TABLE II
ROUND-TRIP CYCLES(RTC)

| | Big core | | Little core | |
|---|---|---|---|---|
| | RTC | Stdev | RTC | Stdev |
| Function call | 166 | 0.00 | 249 | 6.62 |
| Data update (w/o check) | 57 | 0.30 | 30 | 3.58 |

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 data update. In each operation, we count the cycle count of 100 empty function calls and the difference between the cycle count of 100 normal data updates and 100 wrapped data updates. Each operation is repeated 10 times for accurate results. The average results are reported in Table II, which shows the efficiency of our trampoline implementation. As a comparison, the round-trip cycle of a hypervisor call (typically used for context switching in ARM virtualization) is over 500 on Cortex-A53 cores according to [24].

### 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 measure such overhead using the LMBench test suite. Table III reports the results of the experiments, which indicates that the 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 achieved 281K IOPS, and the isolated drivers achieved 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 achieved 118K IOPS, and the isolated drivers achieved 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 achieved 1238K IOPS (92.4% of the non-isolated performance) and 1151K IOPS (85.9% 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.

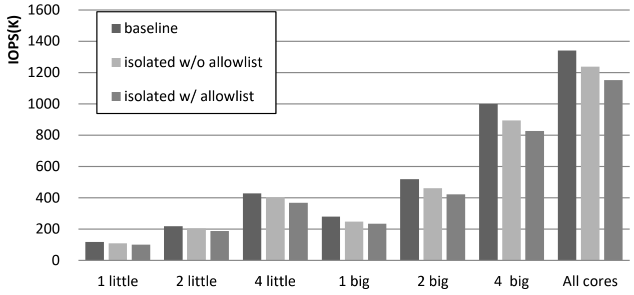

Fig. 3. Performance comparation of the non-isolated and isolated `dummy` drivers.

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 [10], an isolation solution based on PAC and MTE, estimated a 20% maximum performance degradation in its experiments with `ipv6.ko`. The LVDs solution [8], 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 introduced 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** Currently, our prototype only supports one isolation domain for kernel drivers due to the limited watchpoints (only four) in our development board. However, DriverJar could scale to multiple domain isolation with more watchpoints provided. Since the maximum number of watchpoints can be up to 16, there could be at most 12 isolation domains supported with a straightforward one-watchpoint-for-one-driver design. Furthermore, we regard exploring more scalable lightweight driver isolation solutions for ARM devices as our future work.

**Porting efforts** During the driver porting progress, we realize that generating and replacing wrappers for kernel functions called by device drivers can be largely automated. To facilitate pointer replacement and cross-domain data update wrapping, we implemented a source code analysis tool based on libclang for identifying control and data flows. However, due to the limited accuracy of our tool, 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. Exploring fully automatic porting support for legacy drivers is regarded as our future work.

**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, in our current implementation, the driver stack is coupled with a thread as long as the thread invokes the isolated driver and is not freed until the thread exiting. This brings inefficient memory utilization. In the future, it could be reduced by preparing a per-driver stack pool and allocating the driver stack to threads on demand.

TABLE III
LMBENCH RESULTS (IN $\mu$S).

| | Big core | | | | Little core | | | |
|---|---|---|---|---|---|---|---|---|
| | Baseline | Modified | Stdev | Overhead | Baseline | Modified | Stdev | Overhead |
| null syscall | 0.19 | 0.19 | 0.000 | 1.00x | 0.29 | 0.28 | 0.005 | 0.99x |
| open/close | 2.65 | 2.62 | 0.020 | 0.99x | 5.82 | 5.87 | 0.034 | 1.01x |
| stat | 1.37 | 1.37 | 0.006 | 1.00x | 2.58 | 2.61 | 0.015 | 1.01x |
| sig. handler inst | 0.43 | 0.43 | 0.000 | 1.00x | 0.63 | 0.64 | 0.009 | 1.01x |
| sig. handler hndl | 8.83 | 8.91 | 0.005 | 1.01x | 11.99 | 11.90 | 0.000 | 0.99x |
| fork+exit | 124.50 | 129.40 | 2.059 | 1.04x | 268.10 | 281.50 | 5.971 | 1.05x |
| fork+execv | 350.00 | 355.20 | 4.750 | 1.01x | 735.40 | 775.00 | 24.900 | 1.05x |
| /bin/sh -c | 942.30 | 963.70 | 5.728 | 1.02x | 2244.80 | 2391.50 | 29.104 | 1.07x |
| page fault | 0.43 | 0.43 | 0.006 | 1.00x | 1.43 | 1.45 | 0.012 | 1.01x |
| mmap | 670.00 | 672.20 | 2.182 | 1.00x | 1631.60 | 1682.10 | 16.736 | 1.03x |

## IX. CONCLUSION

We propose DriverJar, a lightweight device driver isolation solution for ARM devices. DriverJar utilizes hardware watchpoint, a commonly-available feature on off-the-shelf devices, to prevent data of the rest kernel from being corrupted by an adversary through driver vulnerability. To prevent the protection from being bypassed or compromised, we adopted several security designs to harden DriverJar. We implemented a prototype on AArch64 Linux 5.4.117. The performance evaluation shows that DriverJar causes trivial loss in system performance and small overhead to the isolated driver.

## ACKNOWLEGMENTS

The authors are partially supported by the National Key R&D Program of China (No. 2022YFE0113200), the National Natural Science Foundation of China (NSFC) under Grant U21A20464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

## REFERENCES

[1] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, 2011, pp. 1–5.

[2] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, "Nooks: An architecture for reliable device drivers," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, 2002, pp. 102–107.

[3] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the 22nd ACM symposium on Operating systems principles (SOSP)*, 2009, pp. 45–58.

[4] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with api integrity and multi-principal modules," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 115–128.

[5] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in linux," in *Proceedings of 2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[6] W. Qiang, K. Zhang, and H. Jin, "Reducing tcb of linux kernel using user-space device driver," in *Proceedings of 2016 International Conference on Algorithms and Architectures for Parallel Processing*, 2016, pp. 572–585.

[7] V. Narayanan, A. Balasubramanian, C. Jacobsen, S. Spall, S. Bauer, M. Quigley, A. Hussain, A. Younis, J. Shen, M. Bhattacharyya *et al.*, "Lxds: Towards isolation of kernel subsystems," in *Proceedings of 2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 269–284.

[8] V. Narayanan, Y. Huang, G. Tan, T. Jaeger, and A. Burtsev, "Lightweight kernel isolation with virtualization and vm functions," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2020, p. 157–171.

[9] V. J. M. Manès, D. Jang, C. Ryu, and B. B. Kang, "Domain isolated kernel: A lightweight sandbox for untrusted kernel extensions," *Computers & Security*, vol. 74, pp. 130–143, 2018.

[10] D. McKee, Y. Giannaris, C. Ortega, H. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakcs," in *Proceedings of Network and Distributed System Security Symposium 2022 (NDSS)*, 2022.

[11] "HiKey970 - 96Boards(November 2022)," https://www.96boards.org/product/hikey970/.

[12] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines." in *Proceedings of 2004 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 4, no. 19, 2004, pp. 17–30.

[13] R. Nikolaev and G. Back, "Virtuos: An operating system with kernel virtualization," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 116–132.

[14] Y. Sun and T.-c. Chiueh, "Side: Isolated and efficient execution of unmodified device drivers," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[15] A. Srivastava and J. T. Giffin, "Efficient monitoring of untrusted kernel-mode execution." in *Proceedings of 2011 Network and Distributed System Security Symposium (NDSS)*, 2011.

[16] Qualcomm, "Pointer Authentication on ARMv8.3 - Design and Analysis of the New Software Security Instructions (November 2022)," https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[17] ARM, "Developments in the ARM A-Profile Architecture: Armv8.6-A (November 2022)," https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-architecture-developments-armv8-6-a.

[18] J. Jang and B. B. Kang, "In-process memory isolation using hardware watchpoint," in *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[19] J. Jang and B. B. Kang, "Revisiting the arm debug facility for os kernel security," in *Proceedings of the 6th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[20] J. Jang and B. B. Kang , "Selmon: reinforcing mobile device security with self-protected trust anchor," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 135–147.

[21] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "Erim: Secure, efficient in-process isolation with protection keys (mpk)," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1221–1238.

[22] J. Gu, X. Wu, W. Li, N. Liu, Z. Mi, Y. Xia, and H. Chen, "Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 401–417.

[23] "libclang: C Interface to Clang(November 2022)," https://clang.llvm.org/doxygen/group__CINDEX.html.

[24] D. Kwon, H. Yi, Y. Cho, and Y. Paek, "Safe and efficient implementation of a security system on arm using intra-level privilege separation," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pp. 1–30, 2019.