Lu K, Wang PF, Li G *et al.* Untrusted hardware causes double-fetch problems in the I/O memory. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 33(3): 587–602 May 2018. DOI 10.1007/s11390-018-1842-3

Untrusted Hardware Causes Double-Fetch Problems in the I/O Memory

Kai Lu[1,2,3], *Member*, *CCF*, Peng-Fei Wang[1,*], *Member*, *CCF*, Gen Li[1], and Xu Zhou[1], *Member*, *CCF*

[1] *College of Computer, National University of Defense Technology, Changsha 410073, China*

[2] *Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology Changsha 410073, China*

[3] *Collaborative Innovation Center of High-Performance Computing, National University of Defense Technology Changsha 410073, China*

E-mail: {kailu, pfwang}@nudt.edu.cn; superligen@gmail.com; zhouxu@nudt.edu.cn

Received July 1, 2017; revised February 13, 2018.

Abstract The double fetch problem occurs when the data is maliciously changed between two kernel reads of the supposedly same data, which can cause serious security problems in the kernel. Previous research focused on the double fetches between the kernel and user applications. In this paper, we present the first dedicated study of the double fetch problem between the kernel and peripheral devices (aka. the hardware double fetch). Operating systems communicate with peripheral devices by reading from and writing to the device mapped I/O (input and output) memory. Owing to the lack of effective validation of the attached hardware, compromised hardware could flip the data between two reads of the same I/O memory address, causing a double fetch problem. We propose a static pattern-matching approach to identify the hardware double fetches from the Linux kernel. Our approach can analyze the entire kernel without relying on the corresponding hardware. The results are categorized and each category is analyzed using case studies to discuss the possibility of causing bugs. We also find four previously unknown double-fetch vulnerabilities, which have been confirmed and fixed after reporting them to the maintainers.

Keywords hardware double fetch, double-fetch bug, I/O memory, peripheral device, double-fetch vulnerability

1 Introduction

Hardware is the fundamental component of a computer system. The software works based on the functioning of the underlying hardware, which provides the basic computing, data transmission, and interaction with users. Therefore, hardware has the "privilege" that can access and manipulate data prior to the software. However, "with great power comes great responsibility," the reliability of hardware has always been a critical issue to the computer security.

The newly emerged hardware-based attacks in recent years have placed attention on hardware and peripheral devices. One of the most successful hardware-based attacks is conducted on the USB (universal serial bus) device, which includes most of the computer peripherals, such as disk drives, keyboards, and cameras. These attacks are facilitated by the fact that USB-based devices can operate by plugging without a system restart. For example, the USB autoplay attack[1] enables attackers to run executables by simply plugging a compromised USB device in the computer. Moreover, the trend of IoT (Internet of Things) makes hardware security more important. Billions of everyday objects① such as vehicles, electronic devices, embedded sensors, and other physical hardware are connected to the Internet, and security failures of these endpoints can cause immediate risks of privacy, savings, well-being, or even lives.

Regular Paper

The work is supported by the National Key Research and Development Program of China under Grant No. 2016YFB0200401.

*Corresponding Author

①Eddy N. Gartner: 21 billion IoT devices to invade by 2020. https://www.informationweek.com/mobile/mobile-devices/gartner-21-billion-iot-devices-to-invade-by-2020/d/d-id/1323081, Feb. 2018.

©2018 Springer Science + Business Media, LLC & Science Press, China

The risks are not only in what the devices carry but also built into the core of how they work. In Blackhat 2014, a newly emerged approach called BadUSB②performed attacks by writing malicious code onto USB control chips (also known as firmware) used in thumb drives and smartphones. Anti-virus programs are only designed to scan for software written onto the memory rather than the firmware, and thus, they are unable to detect the infection③. With such compromised hardware, an attacker could completely take over a PC (personal computer), invisibly alter files installed from the memory stick, or redirect the user's Internet traffic, all without being detected④.

The underlying issue is the inability to guarantee and verify the functionality and integrity of the connected devices. The operating system verifies a device through the code on the device⑤; however, in practice, the operating system is unable to ascertain whether a device is normal or compromised since the device might be controlled and reprogramed. When a compromised device is connected to the computer, the driver, which is usually designed without considering the risks posed by compromised devices, is unable to notice the difference. Therefore, malicious data crafted by an attacker can disrupt the kernel via such compromised devices and cause security-related problems, including the double fetch problem.

A double fetch problem is a data inconsistency problem caused by an unexpected data change between two kernel reads of the supposedly same data. When the kernel reads twice the data from the same location, usually the first read is to verify the data and the second read is to use it, under the assumption that the data is unchanged. However, when this assumption is violated by an unexpected data change, problems such as buffer overflows, information leakage, and kernel crashes[2] can be caused in the kernel. Previous work[3-5] only focus on the double fetch problem between the kernel and the user processes, in which the data is changed by a concurrently running user thread under race conditions. However, a double fetch problem can also occur in the I/O memory between the kernel and the peripheral devices (aka., the hardware double fetch). The operating system controls a peripheral device via writing and reading of its registers. When the communication adopts the memory-mapped I/O, reading from and writing to the I/O memory work in exactly the same way when accessing the regular memory. Since the data in the I/O memory is mapped from the external devices, when a compromised device is connected and performs malicious data changes between the two kernel reads of the same I/O memory data, a hardware double fetch problem occurs.

In this paper, we present a dedicated study on the hardware double fetch problem. Using a static pattern-matching approach, we conduct an investigation on the Linux kernel with the aim of addressing the following questions. "How is peripheral data used by the kernel?" "Do hardware double fetches really cause problems?" "In what ways do the kernel fetches cause a hardware double fetch?" "What are the consequences of hardware double fetches?" In summary, we make the following contributions.

• We present the first study (to the best of our knowledge) on the hardware double fetch problem, which provides a new perspective to the double fetch problem by increasing the scope to include the peripheral devices.

• We devise a tool based on static pattern-matching to find hardware double fetches in the Linux kernel. It can analyze all the driver code in one execution without relying on the corresponding hardware. We have made it publicly available online for future research.

• We identify 361 occurrences of hardware double fetches from Linux kernel 4.10.1, including four previously unknown double-fetch vulnerabilities. We report the vulnerabilities to the maintainers and provide patches for them. All of the vulnerabilities have been confirmed and fixed as a result of our report.

• We conduct a thorough investigation of the cases we identify. The cases are categorized by patterns and each category is analyzed with examples to discuss the possibility of causing bugs. We also summarize findings based on our investigation.

The rest of the paper is organized as follows. Section 2 presents relevant background on the I/O memory in Linux and the double fetch problem. Section 3 introduces the static pattern-matching approach we pro-

②Karsten N, Jakob L. BadUSB — On accessories that turn evil. https://pacsec.jp/psj14/PSJ2014_Karsten_Nohl_141112.BadUSB-Pacsec.KN01.pdf, Feb. 2018.

③Finkle J. Hackers can tap USB devices in new attacks. http://www.reuters.com/article/us-cybersecurity-usb-attack-idUSKBN0G00K420140731, Feb. 2018.

④Greenberg A. Why the security of USB is fundamentally broken. https://www.wired.com/2014/07/usb-security/, Feb. 2018.

⑤Paganini P. How to transform USB sticks into undetectable malicious devices. http://securityaffairs.co/wordpress/28855/-hacking/usb-attack-code-released.html, Feb. 2018.

pose and presents the results we obtain. Section 4 presents the analysis of the results, which is based on the case study of different categories. Section 5 discusses the findings, the evaluations of our approach, and the strategies on preventing hardware double fetches. Section 6 is related work, followed by conclusions in Section 7.

2 Background

We provide readers with a reminder of how data is exchanged between the operating system and the peripheral devices in Linux, how drivers work to bridge the hardware and the software, and how double fetch problems happen.

2.1 Memory-Mapped I/O vs Port-Mapped I/O

Every peripheral device is controlled by writing to and reading from its registers. Most of the time, a device has several registers, and they are accessed at consecutive addresses, either in the memory address space or in the I/O address space[6]. Microprocessors from the system normally use two complementary methods to connect peripheral devices: memory-mapped I/O and port-mapped I/O.

Memory-mapped I/O (MMIO) uses the same address space to address both the physical memory and the I/O devices, and the areas of the addresses used by the CPU (central processing unit) must be reserved for I/O. The CPU uses the same instructions used to access the physical memory to access device registers and device memory. Each I/O device monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the data bus to the desired device's hardware register[7].

Port-mapped I/O (PMIO) is also called the isolated I/O. It uses a separate, dedicated address space (accomplished by an extra "I/O" pin on the CPU or an entire bus dedicated to I/O) and is accessed via a set of CPU instructions designed for performing I/O, such as the `in` and `out` instructions (e.g., `inb`, `outb`, `inw`, and `outw`) on the x86 and x86-64 architectures, which copy bytes between the EAX register and a specified I/O port assigned to the I/O device[7].

MMIO uses regular instructions to access physical memory as well as to manipulate an I/O device, which allows a CPU discarding the extra complexity that PMIO brings and requires less internal logic than PMIO. Thus it is cheaper, faster, and easier to build, consumes less power, and can be physically smaller. Besides, all the CPU's addressing modes are available for the I/O and the memory. Instructions that perform an ALU (arithmetic logic unit) operation directly on a memory operand can be used with I/O device registers as well[7].

PMIO instructions need dedicated instructions to access I/O devices, which are often very limited and provided only for simple load-and-store operations between CPU registers and I/O ports[7]. However, PMIO needs less logic than MMIO to decode a discrete address; thus it costs less to add hardware devices to a machine, whereas MMIO must fully decode the entire address bus for every device[8]. Besides, the I/O operations can slow memory access as the address and data buses are shared. This is because the peripheral device is usually much slower than the main memory[7].

Although the use of PMIO is common for ISA (instruction set architecture) peripheral boards[6], most PCI (peripheral component interconnect) devices, especially in x86-based architectures, prefer MMIO because the instructions that perform PMIO are limited to only one register (EAX)[7]. Besides, advantages such as the independence on special-purpose processor instructions, the high CPU efficiency on memory access, and the freedom of compiler in register allocation and addressing-mode selection when accessing memory[6], all make MMIO preferable in the communication between modern peripheral devices and the CPU.

Both memory-mapped registers and memory-mapped device memory are called I/O memory because the difference between registers and memory from peripheral devices is transparent to software[9]. Device memory plays a significant role in exchanging large blocks of data, such as video data and Ethernet packets, but device registers have more complex functionalities, which are generally divided into three types.

- Status registers provide status information of the device to the CPU. These registers are often read-only.
- Configuration/control registers are used by the

[6] I/O ports and I/O memory. http://www.makelinux.net/ldd3/chp-9-sect-1, Feb. 2018.

[7] Memory-mapped I/O. https://en.wikipedia.org/wiki/Memory-mapped_I/O, Feb. 2018.

[8] MEMORY-MAPPED I/O VS PORT-MAPPED I/O. http://www.bogotobogo.com/Embedded/memory_mapped_io_vs_port_mapped_isolated_io.php, Feb. 2018.

[9] Using I/O memory. http://www.makelinux.net/ldd3/chp-9-sect-4, Feb. 2018.

CPU to configure and control the device. Most bits in control registers can be both read and written.

• Data registers are used to read data from or send data to the I/O device. They can be both read and written.

In some instances, a given register may fit more than one of the above categories, e.g., some bits are used for configuration while other bits in the same register provide status information[10].

One distinctive feature of the I/O memory is that the data mapped in the I/O memory can be changed by the peripheral device. For example, the data in a status register changes with the running status of the device, and the data in a data register can be modified by the device as well. Thus, the I/O memory is different from the regular memory, though both being accessed in a similar fashion. The data that resides in the regular memory does not change unless an abnormal situation occurs, such as a race condition (the traditional double fetch). However, in the I/O memory, it is normal that the data changes with the running of the mapped device, making it complicated to identify the hardware double fetches.

Therefore, identifying hardware double fetches from the I/O memory and analyzing the possibility of causing bugs should take I/O memory characteristics (e.g., the data comes from different peripheral sources, mapped data changes with the hardware) into consideration.

2.2 Memory Access in Drivers

Device drivers are critical kernel-level programs that bridge the hardware and the software by providing interfaces between the operating system and the attached devices. Drivers are a large part of current operating systems, e.g., 44% of the Linux source files belong to drivers. However, device drivers are also found to be particularly bug-prone kernel components. According to an empirical study by Chou *et al.*[6], the error rate in device drivers is about 10 times higher than that in any other parts of the kernel. Ten years later, Palix *et al.* replayed Chou *et al.*'s experiment on new Linux versions and found that drivers still have the largest number of faults in absolute terms[7]. Swift *et al.*[8] also found that 85% of system crashes in Windows XP can be blamed on driver errors.

Device drivers are responsible for enabling the kernel to communicate with and make use of the devices connected to the system. For example, in the case of memory-mapped I/O communication, I/O memory regions must be allocated prior to use (e.g., `request_mem_region()` in Linux). Then the I/O memory should be made accessible to the kernel by setting up the mapping, which is done by `ioremap()` to assign virtual addresses to I/O memory regions. Finally, the I/O memory is accessed via a set of wrapper functions (e.g., `ioread8()`) which are optimized and safe to conduct the read and write operations. All the above operations are done in the driver. However, a major problem of the drivers in this procedure is the inability to validate the devices attached to the system. A driver is usually designed to identify the corresponding device by reading the code from the device, which is a rather simple scheme that cannot be used to effectively distinguish a compromised device. A compromised and controlled device could pass the check (if any) at the beginning but perform destructive activities afterward; however, the driver processes it as normal without considering the risk posed by it.

The widely-adopted dynamic approaches in program analysis and bug detection are based on the execution of the program. Since the execution of driver code relies on the functioning of the undertaking hardware, dynamically detecting the hardware double fetches is unworkable if the corresponding hardware is absent. Moreover, the Linux kernel now supports 23 hardware architectures[7], and it is infeasible to have all the hardware (architectures) at one time to give a thorough analysis of the whole kernel including all drivers. Therefore, the dynamic analysis used by the prior double fetch research[3-4] is not a viable approach for a thorough analysis of the hardware double fetches. We propose a static pattern-matching approach to identify hardware double fetches in the Linux kernel including the complete space of drivers.

2.3 Traditional Double Fetch

Generally, a double fetch occurs in the memory accesses between the kernel and the user space. This term was first introduced by Serna[11]. Jurczyk and Coldwind[3] presented the first approach on double fetch detection based on dynamically memory access

[10]Memory-mapped I/O. http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch14s03.html, Feb. 2018.

[11]Serna F J. MS08-061: The case of the kernel mode double-fetch. https://blogs.technet.microsoft.com/srd/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/, Feb. 2018.

tracing, known as the Bochspwn project. Technically, a double fetch takes place within a kernel function, such as a syscall, which is invoked by a user application from user space. As illustrated in Fig.1, the kernel function fetches a value twice from the same memory location in the user space. The first time is to verify it and the second time is to use it. Meanwhile, within the time window between the two kernel fetches, a concurrently running user thread changes the value under a race condition. Then, when the kernel function fetches the value a second time to use, it gets a different value, which may not only cause a difference in computation but also cause consequences such as buffer overflows, information leakage, and privilege escalations[2].

The double fetch problem is similar to the TOCTTOU (time-of-check to time-of-use)[9] issue which is also caused by changes between the check of a condition and the use of the result of that check (by which the condition no longer holds). However, a double fetch is different from a typical TOCTTOU issue because a double fetch focuses on the interaction across a system "boundary", such as across the kernel and user space, whereas a TOCTTOU usually occurs within the same domain. In addition, a TOCTTOU is often specific to a shared object (e.g., a file or a socket)[10–15].

In order to make a clear distinction with the hardware double fetch problem we propose, in this paper, we use term traditional double fetch to describe the double fetches occur between the kernel and the user space. We use term double fetch to represent both the traditional double fetch and the hardware double fetch.

2.4 Hardware Double Fetch

The double fetch problem is essentially a data inconsistency problem that disrupts the kernel, which is not necessarily caused by race conditions in multithreading. Another possibility is by the change of the peripheral data in the I/O memory, namely, a hardware double fetch.

As Fig.2 shows, a hardware double fetch occurs when the kernel controls a peripheral device by reading from and writing to its registers or accessing the device memory data via the memory-mapped I/O. The kernel reads the "same" I/O memory data twice, assuming the data is unchanged. However, since the driver is unable to fully validate the attached device, the compromised

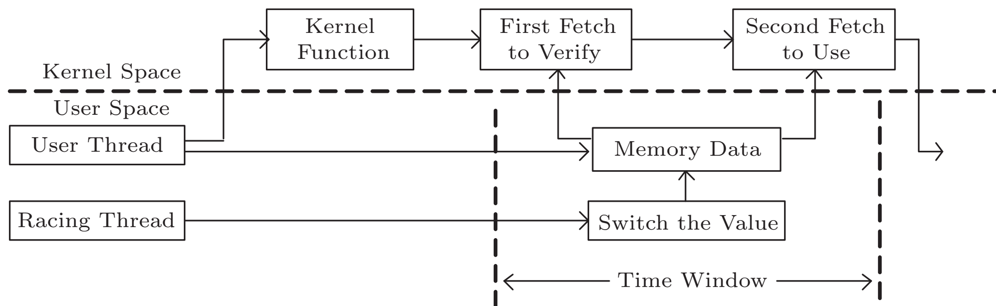

Fig.1. Illustration of how the traditional double fetch occurs.

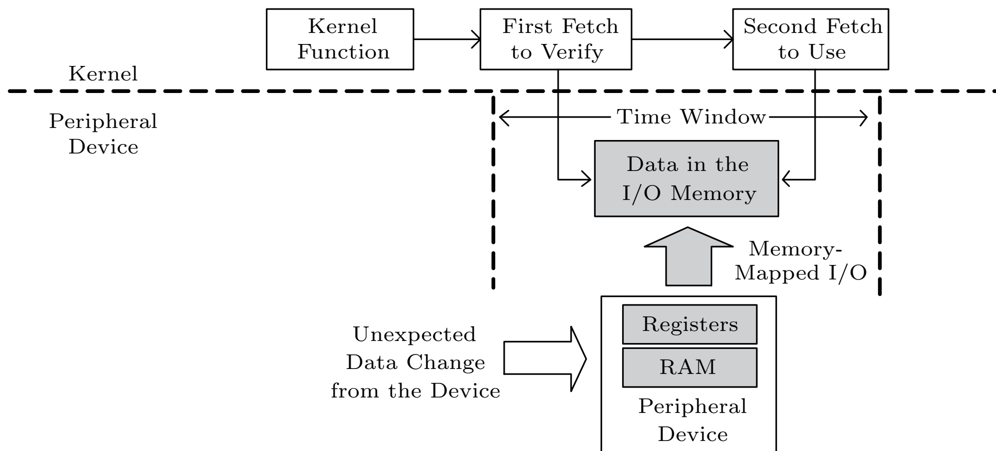

Fig.2. Illustration of how the hardware double fetch occurs.

hardware could tamper the peripheral data mapped in the I/O memory between the two reads, causing the data inconsistency for the kernel use, which may lead to serious problems for the kernel functioning or even a security vulnerability.

A hardware double fetch is similar to a traditional double fetch because the data in both cases is assumed unchanged by the kernel, while it could have been changed by a malicious operation under certain situations. Nevertheless, a hardware double fetch is different from a traditional double fetch in the following aspects.

1) The occurrence crosses different system boundaries. A traditional double fetch occurs between the kernel and the user space, whereas a hardware double fetch occurs between the kernel and the peripheral devices. The involvement of hardware makes the hardware double fetch a complicated problem that no dedicated research was conducted on before.

2) The root cause is different. In a traditional double fetch, the kernel fetches data from user space and the data is modified by the user thread under race conditions. But in the hardware double fetch, the kernel fetches the data from the I/O memory mapped from the peripheral hardware and the data is modified by the peripheral hardware.

3) The peripheral data is inconstant. As discussed in Subsection 2.1, the peripheral data mapped in the I/O memory changes with the running status of the device, which is different from the regular memory whose data does not change. This makes it difficult to identify a hardware double fetch by dynamically checking the fetched value, because even though we find that two kernel fetches get different values from the same I/O memory location, we cannot tell whether the difference is caused by the normal hardware change or abnormal data tamper.

4) Previous approaches are not workable. As discussed in Subsection 2.2, the involvement of the hardware makes the dynamic approaches[3-4] that were used in the prior traditional double fetch research not viable for the hardware double fetches. Since dynamic approaches rely on corresponding hardware to execute the program, it is infeasible to have all the hardware and architectures at one time to conduct a thorough analysis of the entire kernel (including all the drivers).

Therefore, we present this dedicated study to the hardware double fetch problem, aiming to raise its awareness. For the avoidance of ambiguity, we use the term hardware double fetch or the hardware double-fetch situation to represent the situation that the kernel fetches the supposedly same data twice from the I/O memory, which is not necessarily buggy. We use the term buggy hardware double fetch or the hardware double-fetch bug to represent the situation that can actually cause buggy results.

3 Static Pattern-Matching

In this section, we provide details of our static pattern-matching approach and describe how we identify the hardware double fetches from the Linux kernel.

3.1 Overview

We develop a static pattern-matching approach, which can effectively identify the hardware double fetches from the complete Linux kernel. As Fig.3 shows, our approach works directly on the source code of the Linux kernel, and the whole procedure is divided into four stages.

1) *Identify*. The double fetch problem has a clear pattern that the kernel reads from the same memory address twice in the same context. For a hardware double fetch, it is similar except that the kernel reads peripheral data from the I/O memory. In Linux, reading from and writing to the I/O memory are undertaken by dedicated wrapper functions (listed in Table 1), and these functions are the only way to access the peripheral data in the I/O memory. Therefore, we identify the consecutive invocations of wrapper functions read the data from the same I/O memory address. We keep source files that match this pattern and abandon the rest.

2) *Switch*. Since there are dozens of wrapper functions provided by the kernel (as listed in Table 1) to read from and write to the I/O memory, a hardware double fetch could involve any of them. If we take all of these wrapper functions into consideration, there would be hundreds of combinatorial situations to match, which is unworkable. Therefore, we adopt a unified method by using one function to represent the similar wrapper functions, which facilitates the matching without affecting the result. We use `read_wrapper()` to replace the wrapper functions in rows No.1~No.4 in Table 1, use `write_wrapper()` to replace wrapper functions in rows No.5~No.8, use `block_read_wrapper()` to replace wrapper functions in row No.9, and use `block_write_wrapper()` to replace wrapper functions in row No.10. This switch dramatically reduces the situations we need to consider.

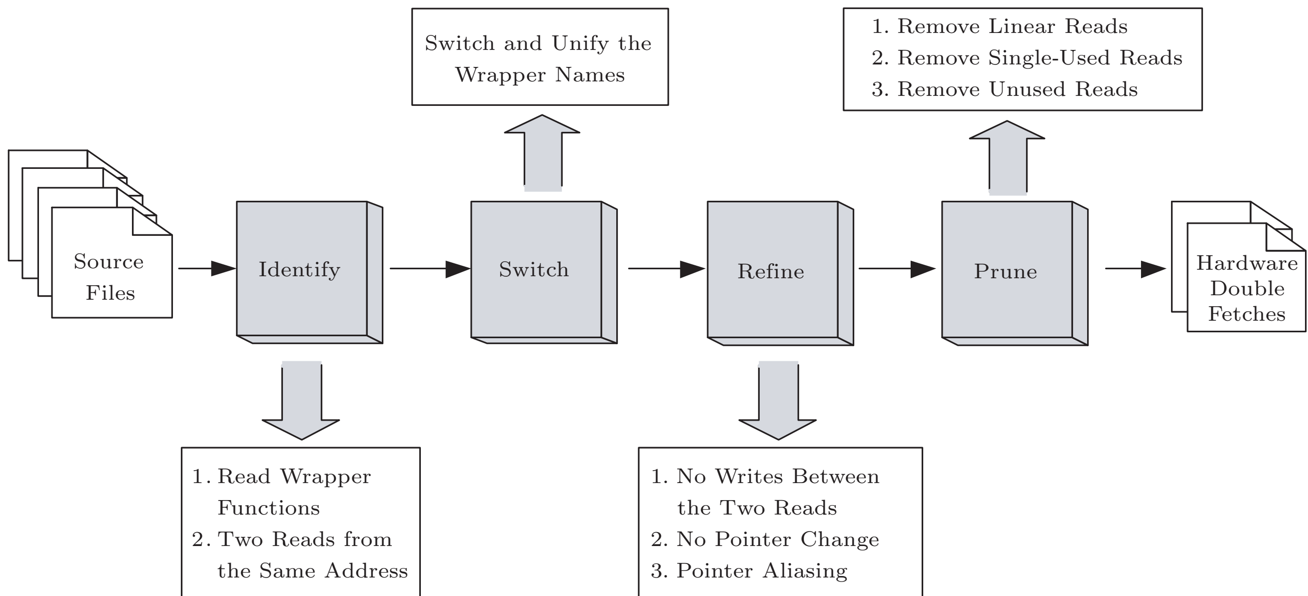

Fig.3. Overview of the static pattern-matching approach.

Table 1. Wrapper Functions in the Linux Kernel

No.	Wrapper Functions to Access I/O Memory	Description
1	`ioread8(s)`, `readb(s)`, `__raw_readb(s)`	Read 8 bits (1 byte) from `s`
2	`ioread16(s)`, `readw(s)`, `__raw_readw(s)`	Read 16 bits (1 word) from `s`
3	`ioread32(s)`, `readl(s)`, `__raw_readl(s)`	Read 32 bits (1 dword) from `s`
4	`ioread64(s)`, `readq(s)`, `__raw_readq(s)`	Read 64 bits (1 qword) from `s`
5	`iowrite8(v,d)`, `writeb(v,d)`, `__raw_writeb(v,d)`	Write 8 bits (1 byte) `v` to `d`
6	`iowrite16(v,d)`, `writew(v,d)`, `__raw_writew(v,d)`	Write 16 bits (1 word) `v` to `d`
7	`iowrite32(v,d)`, `writel(v,d)`, `__raw_writel(v,d)`	Write 32 bits (1 dword) `v` to `d`
8	`iowrite64(v,d)`, `writeq(v,d)`, `__raw_writeq(v,d)`	Write 64 bits (1 qword) `v` to `d`
9	`ioread8_rep(d,s,c)`, `ioread16_rep(d,s,c)`, `ioread32_rep(d,s,c)`, `ioread64_rep(d,s,c)`, `memcpy_fromio(d,s,c)`	Repetitively read `v` (byte, word, dword, qword) from `s` to `d`
10	`iowrite8_rep(d,s,c)`, `iowrite16_rep(d,s,c)`, `iowrite32_rep(d,s,c)`, `iowrite64_rep(d,s,c)`, `memcpy_toio(d,s,c)`	Repetitively write `v` (byte, word, dword, qword) from `s` to `d`

3) *Refine*. In this stage, we consider the following factors that affect the occurrences of hardware double fetches.

• *No Writes Between Two Reads*. In a hardware double fetch, we match the pattern that the kernel reads data from the same address twice, which implies the kernel fetches the same data, i.e., there is the dependence between the two fetches. Therefore, the double-fetched data should not be overwritten by the kernel itself because an additional write will violate the dependence between the two fetches, making the kernel not double-fetch the same data.

• *Pointer Aliasing*. A hardware double fetch reads data from the same memory location but not necessarily uses the same pointer. Sometimes the kernel checks the data via one pointer but uses the data via another pointer. This is more convenient because the original pointer might have been changed in the first read (e.g., processing long messages section by section in a loop). We currently only consider situations with one aliasing pointer (aka. only one pointer assignment occurs either before the first read or between the two reads), which covers most of the cases in the double-fetch problem.

• *Pointer Change*. Even if a hardware double fetch uses the same pointer to fetch data, the pointer is not guaranteed always pointing to the same address because the pointer can be changed between the two fetches. For example, by adding an offset, by self-increment (i.e., ptr++), or by being assigned to a new value, all of these situations will change the pointer. Besides, it is also possible that only a part of the pointer expression is changed, making the address of the pointer also change.

We take such potential pointer changes into consideration to make sure that a hardware double fetch reads from the same address.

4) *Prune*. As a static approach, our approach inevitably introduces false positives, and we try to eliminate them at this stage. We remove the following situations that can cause false reports.

• *Linear Reads*. In some situations, the data transmission is limited by the register bit width of the peripheral device; thus, a value has to be sent separately by linear reads and then linked together. For example, when a 16-bit value is read by the CPU via an 8-bit register, from the view of the code, there will be two reads from the same address. However, the two reads are in fact fetching different data: one read gets the lower 8 bits and the other gets the higher 8 bits. Then the two parts are linked by bit or (i.e., `|`). Thus, the linear reads situation fetches from the same address but gets different sections of the data. This situation should be removed because the whole data is only fetched once.

• *Not Double Used*. A similar situation reads twice the same data from the I/O memory but each time only keeps a part of it. For example, the first read gets the lower 8 bits by `read_wrapper(src) & 0xff`, while the second read gets the higher 8 bits by `read_wrapper(src) >> 8`. This situation should be removed because each read actually gets a different section of the data; thus, no overlapped part is double-used.

• *Unused Reads*. In some situations, a read operation is used to release a signal or to introduce a time delay. The read value is not really returned or used, thus, not causing any problem. These cases should also be removed.

5) *Manual Review*. After the automatic pattern matching, we manually review the reports to confirm the buggy cases. A hardware double-fetch bug satisfies four conditions.

a) The data fetched from the peripheral hardware is critical values, such as struct sizes, buffer lengths, queue pointers, and message sequence numbers. Tampering with such data can lead to severe consequences.

b) The kernel fetches the peripheral data twice from the I/O memory, which gives attackers the chance to tamper the data between the two fetches.

c) The kernel uses the data from both of the two fetches, and thus the data inconsistency can affect the kernel.

d) No additional validation exists after the second fetch; otherwise, tampering with the data can be prevented. No infeasible cases exist in the double-fetch situations, such as pointer changes.

The pattern-matching phase in our approach automatically identifies hardware double-fetch situations that satisfy conditions b) and d). Afterwards, we manually review the reported cases to find the vulnerabilities that also match conditions a) and c). Finally, we report the potential vulnerabilities to the kernel maintainers who make the final confirmation.

Our tool can help to locate the potential hardware double fetches from thousands of source files and remove the infeasible cases. However, the use of the data in the double-fetch problem is complicated. Devising a tool to automatically and accurately track where and how the fetched data is used and gets the same results would cost much more time than our current tool with manual efforts.

3.2 Implementation

Our tool is implemented based on Coccinelle[16], which is a program matching and transformation engine that provides the language SmPL (Semantic Patch Language) for specifying desired matches and transformations in C code. Since Coccinelle's strategy for traversing control-flow graphs is based on temporal logic CTL (computational tree logic)[17], the pattern-matching implemented on Coccinelle is path-sensitive, which achieves better code coverage. Coccinelle is highly optimized to improve the performance when exhaustively traversing all the execution paths. Besides, Coccinelle is insensitive to newlines, spaces, and comments, which achieves better precision. Moreover, the pattern-based analysis is applied directly to the source code; therefore, wrappers that are defined as macros (e.g., `__raw_readb()`) will not be expanded during the matching, which facilitates the matching of fetches in our approach by identifying wrapper functions.

Our implementation is about 2.3 KLOC (thousands of lines of code), which combines the SmPL script to match the pattern as well as transforming the wrapper functions, and the Python script to process the results. We have made it publicly available⑫, hoping that it is useful for future study.

⑫https://github.com/wpengfei/hardware_df, Feb. 2018.

3.3 Results

We apply our static pattern-matching approach to the Linux kernel of version 4.10.1, which was the newest version when the experiment was conducted. After the automatic pattern-matching, we get 178 candidate files out of 42 417 source files (.c or .h files) from the Linux kernel, including 361 hardware double fetches. Then we manually review these files and divide them into five categories according to the types of the I/O memory.

As Table 2 shows, status register checking accounts for most of the hardware double fetches we identify (65.9%), which includes four typical patterns. The common check is a situation that multiple conditional checks of the same variable are used one after another, which has 59 occurrences. The loop check situation uses a loop to constantly check the register value, which has 81 occurrences. The wait check situation uses wait functions (e.g., `udelay()`) between the two fetches, which has 81 occurrences. The stable check situation compares the values from the two fetches, which has 18 occurrences. All of these situations are likely to turn buggy once a malicious data change occurs between the two fetches. However, we do not confirm a currently buggy case.

Table 2. Results of the Categorized Hardware Double Fetches

Type	Pattern	Counts (Percentage)	Number of True Bugs
Status Register	Common check	59 (16.3)	0
	Loop check	80 (22.2)	0
	Wait check	81 (22.4)	0
	Stable check	18 (5.0)	0
Config Register	Configure check	29 (8.0)	0
Data Register	Check and use	68 (18.8)	3
Dev. Mem.	Block check	1 (0.3)	1
Special	Flush write	17 (4.7)	0
	Double valid	6 (1.7)	0
	Delay	2 (0.6)	0
Total	–	361 (100)	4

We identify 29 hardware double fetches from the configuration registers. They read the registers the first time to check the value, then read it again to modify the value, and write it back (the configure check situation). However, we do not think this can cause a double-fetch bug because even if the value is changed between the two fetches, it can hardly harm the kernel except giving wrong configuration information to the device.

We identify 68 hardware double fetches from the data registers. They have the same pattern with the traditional double fetches in which the first fetch is for check and the second one is for use. Although most of them are not necessarily buggy as the double-fetched value is not double-used, we find three buggy cases (which will be discussed in Section 4).

We only find one hardware double fetch in the device memory. Since the device memory is usually used to exchange large blocks of data, developers usually try to avoid double-fetching large blocks of data to improve efficiency. However, this one has been confirmed as a double-fetch vulnerability (which will be discussed in Section 4).

Finally, we also find some hardware double-fetch situations that happen due to the hardware-specific features. For example, some hardware needs an extra read to flush the write down to the hardware (the flush write situation), some hardware requires to be read twice to get the valid value (the double valid situation), and some hardware needs an extra read to delay the real read, while the fetched value of the extra read is useless (the delay situation). These occurrences of double fetches are not likely to cause buggy situations because the data is not double-used.

4 Analysis of Hardware Double Fetches

In this section, we discuss the hardware double fetches we identify. We also describe how they occur in the I/O memory as well as the possibility of causing bugs (and vulnerabilities).

4.1 Status Registers

Status registers indicate the working status of a peripheral device; thus the data in status registers changes with the running of the device. During the communication, the kernel usually checks these registers before each operation to make sure that the device is in the right status, so as to take a right step based on the current status. Therefore, these consecutive reads to the status registers introduce double-fetch situations.

Fig.4 shows a code snippet from file `drivers/parport/parport_ip32.c` in Linux 4.10.1, which illustrates a typical situation we call as the common check. Before each step, `priv->regs.ecr` is fetched and checked (lines 1900, 1929, and 1933), and only when the fetched data satisfies the condition, the program proceeds. However, problems occur if dependence exists among these checks. For example, if one check relies on a previous check and it is assumed that the check still holds while the previous one is not valid anymore due to malicious data changes from the hardware, then unexpected results will be caused.

```
       ⋮
1899  for (i=0; i<1024; i++){
1900     if (readb(priv->regs.ecr) & ECR_F_FULL){
1901        /* FIFO full */
1902        priv->fifo_dept = i;
1903        break;
1904     }
       ⋮
1923  for (i = 0; i < priv->fifo_depth; i++) {
          ⋮
1928     if (!priv->writeIntrThreshold
1929        && readb((priv->regs.ecr) & ECR_SERVINTR)
1930        /* writeIntrThreshold reached */
1931        priv->writeIntrThreshold = i + 1;
1932     if (i + 1 < priv->fifo_depth
1933        && readb(priv->regs.ecr) & ECR_F_EMPTY) {
1934        /* FIFO empty before the last byte? */
1935        pr_probe(p, "Data lost in FIFO\n");
1936        goto fail;
1937     }
1938  }
       ⋮
```

Fig.4. Common check situation in the status registers.

Another typical use of the status register is when the kernel waits for a certain status by constantly checking it in a loop, which we call as the loop check. Fig.5 is a code snippet of such a case from file `drivers/tty/moxa.c` in Linux 4.10.1. The kernel waits `baseAddr + Magic_no` for the desired value `Magic_code`, and checks it in a loop with an interval of 10 ms. This consecutive checking of the same address introduces double-fetch situations. Similar situations also happen with the `while` loop, `do...while` loop, and wait functions such as `udelay()`, `ndelay()`, `mdelay()`, and `cpu_relax()`. Malicious data changes performed by the compromised hardware could disrupt the check, leading to unexpected results.

```
      ⋮
625  for (i=0; i<500; i++){
626     if (readw(baseAddr + Magic_no) == Magic_code)
627        break;
628     msleep(10);
629  }
630  if (readw(baseAddr + Magic_no) != Magic_code)
631     return -EIO
      ⋮
```

Fig.5. Loop check situation in the status registers.

Fig.6 shows a code snippet from file `drivers/clocksource/jcore-pit.c` in Linux 4.10.1, which illustrates a situation we call as the stable check. In this case, the newly fetched data is used to compare with the previously fetched data in a loop until it is equal to one another. This is often used in time-related situations that the data needs to be validated until it is stable.

```
     ⋮
51  do {
52     seclo0 = seclo;
53     nsec = readl(base + REG_NSEC);
54     seclo = readl(base  + REG_SECLO);
55  } while (seclo != seclo)
     ⋮
```

Fig.6. Stable check situation in the status registers.

In summary, the common check situation usually reads status register data for conditional checks one after another. If dependent relations exist among the checks, problems will occur because a previously satisfied condition may have been violated due to an unexpected data change in the registers. For the loop check situation, the use of wait functions expands the time window between the two fetches, which increases the possibility of successfully tampering with the register data. As for the stable check situation, it cannot guarantee the data is stable after the check. Thus, although the hardware double fetches we identify from status registers are not necessarily buggy, they are prone to turning into bugs, once an unexpected data change disrupt the checking.

4.2 Configuration Registers

Configuration (control) registers store the configuration information for the peripheral devices, which directs the working of the devices. Hardware double fetches can be introduced when the driver reads and modifies the information in the control registers.

Fig.7 is from file `/drivers/char/hpet.c` in Linux 4.10.1, which shows a typical situation of how a configuration register is double fetched, and we call it the configure check. Generally, when the driver wants to modify a configuration register, it usually reads the value first to check and make sure whether the device is in the right status (line 638, line 644). Then the value is

```
      ⋮
638  v = readq(&timer->hpet_config);
639  if ((v & Tn_PER_INT_CAP_MASK == 0){
640     err = -ENXIO;
641     break;
642  }
643  if (devp->hd_flags & HPET_PERIODIC &&
644     readq(&timer->hpet_config) & Tn_TYPE_CNF_MASK);
645     v = readq(&timer->hpet_config);
646     v ^= Tn_TYPE_CNF_MASK;
647     writeq(v, &timer->hpet_config);
648  }
      ⋮
```

Fig.7. Hardware double fetch in the configuration registers.

read again (line 645) and modified by changing certain bits (e.g., by `|`, `&`, and `^`). Finally, the modified value is written back to the register (line 47). However, we think this situation is unlikely to cause a double-fetch bug because even if the value is changed between the two fetches, it can hardly harm the kernel except giving wrong configuration information to the device.

4.3 Data Registers

Data registers play a significant role in the communication with the peripheral devices. For example, the data generated during the working of the device is stored in data registers, which are accessible to the kernel. Messages in the communication are often exchanged via data registers (e.g., in the form of commands and responses). Moreover, the data in data registers is often critical and related to memory access, such as the variable that indicates the message length, or the header and tail pointer of a message queue. When such data is compromised, it is very likely to cause memory access errors.

Fig.8 shows a typical situation of how the data register is double-fetched, which we call as the check and use. In file `/drivers/net/can/pch_can.c` of Linux 4.10.1, the driver fetches `mcont` twice from the data register, in which the first time (line 661) is to validate the data (lines 662, 666, 672) and the second time (line 697) is to use the data. Based on the second fetched `mcont`, value `cf->can_dlc` is calculated and used to control the loop at line 700. Since `cf->data[]` is defined as an array of length 8, if `mcont` is unexpectedly changed to a larger value before the second fetch, an array over-access would be caused. Fortunately, macro `get_can_dlc()` at line 697 prevents this by limiting the maximum value of `cf->can_dlc`; thus, it is safe now. However, during the investigation of the code history, we find that this secure macro was introduced in a later patch[13], and before the patching, this code was once vulnerable to this double-fetch bug.

Fig.9 shows another case, which is from file `/sound/isa/msnd/msnd_pinnacle.c` in Linux 4.10.1. Here the data register holds the header pointer `chip->DSPQ+JQS_wHead` of a message queue, and the value is fetched twice at line 178 and line 182, respectively. The first time is to prevent queue over-access in comparison with the tail pointer (the check), whereas the second time is to get the message data and send it by `snd_msnd_eval_dsp_msg()` (the use). This double fetch here is vulnerable because the pointer is stored in the peripheral device register, once the value of the pointer is changed by the device between the check and the use, and an over-boundary access of the message queue would be caused. This hardware double fetch has been confirmed as a vulnerability (CVE-2017-9984) by the maintainers. Similar cases also include CVE-2017-9985 in files `/sound/isa/msnd/msnd_midi.c` and CVE-2017-9986 in file `/sound/oss/msnd_pinnacle.c`.

```
      ⋮
661   reg = ioread32(&priv->regs->ifregs[0].mcont);
662   if (reg & PCH_IF_MCONT_EOB)
663       break;
      ⋮
666   if (reg & PCH_IF_MCONT_MSGLOST) {
          ⋮
672   } else if (!(reg & PCH_IF_MCONT_NEWDAT)) {
          ⋮
675   }
676
677   skb = alloc_can_skb(priv->ndev, &cf);
678   if (!skb) {
679       netdev_err(ndev, "alloc_can_skb Failed\n");
680       return rcv_pkts;
681   }
      ⋮
697   cf->can_dlc = get_can_dlc((ioread32(&priv->regs-> ifregs[0].mcont)) & 0xF);
      ⋮
700   for (i = 0; i < cf->can_dlc; i += 2) {
701       data_reg = ioread16(&priv->regs->ifregs[0].data[i / 2]);
702       cf->data[i] = data_reg;
703       cf->data[i + 1] = data_reg >> 8;
704   }
      ⋮
```

Fig.8. Hardware double fetch in the data register.

```
169 static irqreturn_t snd_msnd_interrupt(int irq, void *dev_id) {
        ⋮
178     while (readw(chip->DSPQ + JQS_wTail)
                    != readw(chip->DSPQ + JQS_wHead)) {
            ⋮
181         snd_msnd_eval_dsp_msg(chip,
182         readw(pwDSPQData + 2 * readw(chip->DSPQ + JQS_wHead)));
183
184         wTmp = readw(chip->DSPQ + JQS_wHead) + 1;
185         if (wTmp > readw(chip->DSPQ + JQS_wSize))
186             writew(0, chip->DSPQ + JQS_wHead);
187         else
188             writew(wTmp, chip->DSPQ + JQS_wHead);
189         }
            ⋮
192         return IRQ_HANDLED;
193 }
```

Fig.9. Hardware double-fetch vulnerability in the data register.

In addition to the above cases, most of the hardware double fetches we identify from the data registers are not buggy because the double-fetched values are not double-used, which is very similar to the traditional double fetches[5].

[13]https://github.com/torvalds/linux/commit/1d5b4b2778e8e40f42ae5d9556777583f3556d81, Feb. 2018.

4.4 Device Memory

In addition to device registers, device memory is also mapped and accessible to the kernel. Since device registers are limited by the bit width, large data blocks (e.g., video data and Ethernet packets) are usually exchanged via mapped device memory for efficiency purpose. The data exchange is usually conducted by block wrapper functions (listed in rows No.9 and No.10 of Table 1).

Fig.10(a) shows such a hardware double fetch in the device memory, which is from `/drivers/media/pci/saa7164/saa7164-bus.c` in Linux 4.10.1. Function `saa7164_bus_get()` uses lines 384 and 385 to fetch device memory data from `bus->m_pdwGetRing` to local buffer `msg_tmp`, and then it uses `msg_tmp` to do a series of checks with the desired data at line 405, including `->size` and `->seqno` fields. After these checks, a second fetch (lines 446 and 447) copies the data from `bus->m_pdwGetRing` to `msg` for later use. This double fetch here is vulnerable because the secondly fetched data in `msg` cannot be guaranteed to satisfy the conditions at line 405. Once the `->size` or `->seqno` field of the secondly fetched message changes, using that a message is likely to cause memory access errors.

In order to prove this assumption, we further trace how this function is called. As Fig.10(b) shows, function `saa7164_cmd_dequeue()` calls `saa7164_bus_get()` at line 161, and the fetched message is stored in `tRsp`. Then, the `seqno` field of `tRsp` is passed to function `saa7164_cmd_free_seqno()` (line 173), within which it is used to access an array (e.g., `dev->cmd[seqno]` at lines 48∼52). However, the message is used without an additional validation, and if the `seqno` field of the message is changed to a very large value before the second fetch, an array over-access error could be caused. This case has been confirmed as a vulnerability (CVE-2017-8831) by the maintainer and fixed already.

5 Discussion

5.1 Findings

In this study, we propose the concept of hardware double fetches and design a static pattern-matching approach to detect them. We identify 361 occurrences in the I/O memory, and based on the investigation of the results, we have the following findings.

1) Hardware double fetches truly exist in the I/O memory and some of them can cause bugs or even vulnerabilities. The occurrences range from all three types (i.e., status, control, and data) of register to the device memory. Among them, we find four previously unknown double-fetch vulnerabilities (listed in Table 3). With these vulnerabilities, attackers can cause over-boundary accesses (buffer overflows) in the kernel by simply changing the value from the device.

2) Status registers commit most of the hardware double fetches we identify (65.9%, 238/361), but data

```
319 int saa7164_bus_get(struct saa7164_dev *dev, struct tmComResInfo* msg,
320                                              void *buf, int peekonly){
      ⋮
384 memcpy_fromio(&msg_tmp, bus->m_pdwGetRing + curr_grp, space_rem);
385 memcpy_fromio((u8 *)&msg_tmp + space_rem, bus->m_pdwGetRing,
                                    bytes_to_read - space_rem);
      ⋮
405 if ((msg_tmp.id != msg->id) || (msg_tmp.command != msg->command) ||
      (msg_tmp.controlselector != msg->controlselector) ||
      (msg_tmp.seqno != msg->seqno) || (msg_tmp.size != msg->size)) {
        /* Check if the command/response matches what is expected */
        ⋮
413     goto out;
414 }
      ⋮
446 memcpy_fromio(msg, bus->m_pdwGetRing + curr_grp, space_rem);
447 memcpy_fromio((u8 *)msg + space_rem, bus->m_pdwGetRing,
448                                 sizeof(*msg) - space_rem);
449 if (buf)
450     memcpy_fromio(buf, bus->m_pdwGetRing + sizeof(*msg) -
451                                              space_rem, buf_size);
    ... }
```

(a)

```
135 static int saa7164_cmd_dequeue(struct saa7164_dev *dev)
136 {
        ⋮
        /* Message is stored in tRsp*/
161     ret = saa7164_bus_get(dev, &tRsp, &tmp, 0);
        ⋮
        /* tRsp.seqno is passed to the following function*/
173     saa7164_cmd_free_seqno(dev, tRsp.seqno);
        ⋮
187 }
-------------------------------------------------------------------
45 static void saa7164_cmd_free_seqno(struct saa7164_dev *dev,
46                                                     u8 seqno){
      ⋮
48    if ((dev->cmds[seqno].inuse == 1) &&
49          (dev->cmds[seqno].seqno == seqno)) {
50                  dev->cmds[seqno].inuse = 0;
51                  dev->cmds[seqno].signalled = 0;
52                  dev->cmds[seqno].timeout = 0;
53          }
      ⋮
55 }
```

(b)

Fig.10. Double-fetch vulnerability in the device memory. (a) How the peripheral data is double-fetched. (b) How the double-fetched data is used.

Table 3. Description of the Identified Double-Fetch Vulnerabilities

ID	File	Description
CVE-2017-8831	Linux-4.10.1/drivers/media/pci/saa7164/saa7164-bus.c	Function `saa7164_bus_get()` allows local users to cause a denial of service (out-of-bounds array access) by changing a certain sequence-number value, aka. a "double fetch" vulnerability
CVE-2017-9984	Linux-4.10.1/sound/isa/msnd/msnd_pinnacle.c	Function `snd_msnd_interrupt()` allows local users to cause a denial of service (over-boundary access) by changing the value of a message queue head pointer between two kernel reads of that value, aka. a "double fetch" vulnerability
CVE-2017-9985	Linux-4.10.1/sound/isa/msnd/msnd_midi.c	Function `snd_msndmidi_input_read()` allows local users to cause a denial of service (over-boundary access) by changing the value of a message queue head pointer between two kernel reads of that value, aka. a "double fetch" vulnerability
CVE-2017-9986	Linux-4.10.1/sound/OSS/msnd_pinnacle.c	Function `intr()` allows local users to cause a denial of service (over-boundary access) by changing the value of a message queue head pointer between two kernel reads of that value, aka. a "double fetch" vulnerability

registers are more likely to cause buggy situations (three have been found) as they hold critical data related to memory access, such as message length variables and queue pointers.

3) Configuration registers are relatively safe as the kernel directs such registers rather than relies on them; thus the corrupted data from a peripheral device can hardly harm the kernel.

4) Device memory introduces the least hardware double fetches (only one) because double-fetching large data blocks is usually avoided by developers for the concern of efficiency.

5.2 Evaluation

5.2.1 *Efficiency*

The automatic pattern-matching process of our approach takes approximately 28 minutes to run a test on the whole Linux kernel of version 4.10.1 including all the drivers. Our approach is much more efficient than the dynamic approaches adopted by the prior traditional double fetch analysis. For example, Bochspwn needs 15 hours to only boot the Windows kernel in their simulator[3].

Although we rely on manual review to confirm the bug, it is also efficient. Manually reviewing the 178 candidate files only takes us a few days, which is acceptable, while building an automatic tool with high precision would take much longer time. Besides, the previous approach also needs to investigate about 200 KB logs generated by the simulator[3]. Furthermore, manually reviewing the source code is helpful in extracting the patterns as well as figuring out why they happen, which is meaningful for a pilot work of such a new topic.

5.2.2 *Effectiveness*

Static approaches inevitably produce false positives due to the lack of runtime information. However, our static approach increases the code coverage without losing too much precision. It successfully identifies 361 hardware double fetches from 42 417 Linux source files. Among them, four are confirmed as vulnerabilities. Therefore, it is effective as a pilot study to prove the concept of hardware double fetches rather than thoroughly detects them. We leave it to the future work to reduce the false reports and improve the precision.

For the false positives, although most of the identified cases are not necessarily buggy, they should not be simply regarded as false positives and removed because some of them have potential risks. For the hardware double fetches in the static registers, one possible way to leverage them is "fuzzing". More specifically, an attacker can use a piece of compromised hardware to simulate the normal hardware and randomly switch the values in the static registers. If dependent relations exist among the checks of the status registers and the value changes right between the two fetches of it, then a hardware double fetch is leveraged. For the hardware double fetches in the data registers, some of them are non-buggy because the value from the hardware is fetched twice but only one of them is used by the kernel; thus, the data change from the hardware is ineffective. Nevertheless, when the code is updated by the developer without paying special attention and the double-fetched values are cross-used (i.e., both values from the two fetches are used), then the change of the data from the hardware can cause data inconsistency for the kernel use and a non-buggy hardware double fetch turns into a buggy one. In such a case, the leverage also relies on manipulating the external hardware or a simulation of it. Although leveraging the hardware double fetches is complicated compared with the traditional double fetches owing to the manipulation of the hardware, it brings high risks as people usually pay less attention to such situations. Thus, the rest cases are still worthy.

As for the false negatives, our approach increases code coverage from two aspects. Firstly, it includes the complete space of drivers for analysis without relying on the specific hardware, which performs better than the dynamic approaches that only cover limited driver code in one execution. Secondly, it is based on the Coccinelle engine, which provides path-sensitive analysis and further improves the code coverage. We do not find any false negatives during the manually checking of the random samples from the source files.

5.2.3 Limitation

Our static pattern-matching approach relies on the source code and cannot analyze the pre-compiled binaries whose source code is unavailable. Therefore, it is limited to the in-house testing for the developers before the drivers are released.

5.3 Prevention

In order to prevent this hardware double-fetch problem, we can take measures from both the software and the hardware.

In the software layer, developers can prevent double-fetch situations by avoiding double-fetching the same data. For double fetches that cannot be avoided, adding validations after each fetch is a simple yet effective way to prevent such bugs. However, additional validations bring extra overhead, and it is not viable for situations such as the consecutive reads in the status registers. Thus, the effectiveness of software-based prevention is limited.

Since the hardware double fetch problem is caused by unexpected data changes from the hardware, if the functionality and the integrity of the connected devices are validated effectively every time when the connection is established, problems are solved. To some extent, researches on trusted hardware[18-19] and tamper-proof hardware[20-21] are helpful. However, these techniques also inevitably cause performance deduction; besides, there is still a long way before these techniques are applied to industrial products. Therefore, hardware validation and tamper-protection still remain as problems, especially in the era of IoT.

5.4 Future Work

We propose the hardware double-fetch problem in this work and identify 361 occurrences, including four vulnerabilities. We extract patterns of how hardware double fetches happen and discuss the possibility of causing bugs. The manual review process in our approach provides useful knowledge to this topic, which is essential to this pilot work. In the future, we will try to develop a more accurate and automatic tool to replace the manual efforts in the bug confirmation process. Besides, we will also pay attention to exploiting such hardware double-fetch vulnerabilities and develop practical schemes to prevent such problems.

6 Related Work

In this study, we propose the problem of hardware double fetch and investigate 361 occurrences identified by our static pattern-matching approach. So far, the most related work to this topic includes the traditional double fetch and the TOCTTOU.

Jurczyk and Coldwind[3] conducted the first systematic work on the double-fetch analysis in their Bochspwn project. It was the first dynamic analysis to detect double fetches based on memory access patterns. They instrumented the Windows kernel to identify double fetches dynamically by observing read operations on the same user space address within a short time window. They found over 100 double fetch instances and some of them are exploitable vulnerabilities. Wilhelm[4] used an approach similar to the Bochspwn project to analyze the memory access pattern of para-virtualized devices' backend components. His analysis identified 39 potential double fetch issues and discovered three novel security vulnerabilities in security-critical backend components. However, their approaches were limited to an analysis of kernel code and drivers for which they had the required hardware, and had severe runtime overhead. In our work, we focus on the hardware double fetch problem, which is similar to the traditional double fetches but occurs in the I/O memory. Besides, we propose a static pattern-matching approach to identify the hardware double fetches, which could analyze all the Linux drivers at one time without relying on the corresponding hardware.

A TOCTTOU issue occurs when a program checks for a particular characteristic of an object, so as to take actions on the assumption that the characteristic still holds, whereas it does not hold any longer[9]. The data inconsistency in TOCTTOU is usually caused by a race condition that results from improper synchronized concurrent accesses to a shared object. There are varieties of shared objects in the computer system, such as files[9], sockets[10], and memory locations[22]. TOCTTOUs often occur in the Unix file system and numerous approaches[11-15] have been proposed to solve

these problems. Yang *et al.*[22] focused on the TOCTTOU problem at memory access level and proposed concurrency attacks by exploiting the time window between the check and the use. However, they did not consider the involvement of kernel. Watson[10] worked on exploiting wrapper concurrency vulnerabilities that come from system call interposition. He also proposed the Time-of-Audit to Time-of-Use and Time-of-Replacement to Time-of-Use issues. None of these prior studies on TOCTTOU takes into consideration of the hardware double fetch problem we propose.

Mulliner and Michéle[23] implemented a "Read-It-Twice" (RIT) attack based on the observation that software installation (and firmware upgrade) assumes that the files on an attached mass-storage device will not change between the check and the install. With an emulated mass-storage device, they switched the content of files to inject a shared object into a Samsung television, which bypassed the security checks and gained root privilege to jailbreak the device. However, RIT targets at the content of files in the storage of the device, whereas the hardware double fetch problem we proposed focuses on the data in the device register and RAM.

Chou *et al.*[6] gave the first thorough study on faults found in Linux, and they found that the error rate in device drivers is about 10 times higher than that in any other parts of the kernel. Ten years later, Palix *et al.*[7] replayed Chou *et al.*'s work on new Linux versions released between 2003 and 2011 with Coccinelle engine, and they pointed out that the kind of faults considered ten years ago was still relevant, and were still present in both new and existing files. Drivers still have the largest number of faults in absolute terms. However, their studies did not pay attention to the double fetch problem in the I/O memory. We conduct the first study on this problem and find vulnerable cases.

7 Conclusions

This work presented the first dedicated study of the double fetch problem in the I/O memory (aka. the hardware double fetch), which provides a new perspective to the double fetch problem by increasing the scope to include peripheral devices. We proposed a static pattern-matching approach to identify the hardware double fetches from the Linux kernel. We categorized the identified 361 occurrences and analyzed each category the possibility of causing bugs. We also found four previously unknown double-fetch vulnerabilities, which have all been confirmed and fixed after reporting them to the maintainers.

Acknowledgment The authors thank the anonymous reviewers for their helpful feedback.

References

[1] Tahir R, Hamid Z, Tahir H. Analysis of AutoPlay feature via the USB flash drives. In *Proc. the World Congress on Engineering*, July 2008.

[2] Wang P F, Lu K, Li G, Zhou X. A survey of the double-fetch vulnerabilities. *Concurrency and Computation Practice and Experience*, 2018, 30(6): e4345.

[3] Jurczyk M, Coldwind G. Identifying and exploiting windows kernel race conditions via memory access patterns. Technical Report, Google Research, 2013. http://pdfs.semanticscholar.org/ca60/2e7193f159a56a3559-f08b677abfba60beb2.pdf, Mar. 2018.

[4] Wilhelm F. Tracing privileged memory accesses to discover software vulnerabilities [Master's Thesis]. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, 2015.

[5] Wang P F, Krinke J, Lu K, Li G, Dodier-Lazaro S. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the Linux kernel. In *Proc. the 26th USENIX Security Symp.*, August 2017.

[6] Chou A, Yang J F, Chelf B, Hallem S, Engler D. An empirical study of operating systems errors. *ACM SIGOPS Operating Systems Review*, 2011, 35(5): 73-88.

[7] Palix N, Thomas G, Saha S, Calvès C, Lawall J, Muller G. Faults in Linux: Ten years later. *ACM SIGPLAN Notices*, 2011, 46(3): 305-318.

[8] Swift M M, Bershad B N, Levy H M. Improving the reliability of commodity operating systems. *ACM Trans. Computer Systems*, 2005, 23(1): 77-110.

[9] Bishop M, Dilger M. Checking for race conditions in file accesses. *Computing Systems*, 1996, 9(2): 131-152.

[10] Watson R N M. Exploiting concurrency vulnerabilities in system call wrappers. In *Proc. the 1st USENIX Workshop on Offensive Technologies*, August 2007.

[11] Chen H, Wagner D. MOPS: An infrastructure for examining security properties of software. In *Proc. the 9th ACM Conf. Computer and Communications Security*, November 2002, pp.235-244.

[12] Cowan C, Beattie S, Wright C, Kroah-Hartman G. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *Proc. the 10th Conf. USENIX Security Symp.*, August 2001, pp.165-176.

[13] Lhee K S, Chapin S J. Detection of file-based race conditions. *International Journal of Information Security*, 2005, 4(1/2): 105-119.

[14] Cai X, Gui Y W, Johnson R. Exploiting Unix file-system races via algorithmic complexity attacks. In *Proc. the 30th IEEE Symp. Security and Privacy*, May 2009, pp.27-20.

[15] Payer M, Gross T R. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Proc. the 8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environments*, March 2012.

[16] Lawall J, Laurie B, Hansen R R, Palix N, Muller G. Finding error handling bugs in OpenSSL using Coccinelle. In *Proc. the 2010 European Dependable Computing Conf.*, April 2010, pp.191-196.

[17] Brunel J, Doligez D, Hansen R R, Lawall J L, Muller G. A foundation for flow-based program matching: Using temporal logic and model checking. In *Proc. the 36th Annual ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, January 2009.

[18] Lie D, Thekkath C A, Horowitz M. Implementing an untrusted operating system on trusted hardware. *ACM SIGOPS Operating Systems Review*, 2003, 37(5): 178-192.

[19] Irvine C E, Levitt K. Trusted hardware: Can it be trustworthy? In *Proc. the 44th ACM/IEEE Design Automation Conf.*, June 2007.

[20] Katz J. Universally composable multi-party computation using tamper-proof hardware. In *Proc. the 26th Annual Int. Conf. the Theory and Applications of Cryptographic Techniques*, May 2007, pp.115-128.

[21] Chandran N, Goyal V, Sahai A. New constructions for UC secure computation using tamper-proof hardware. In *Proc. the 27th Annual Int. Conf. the Theory and Applications of Cryptographic Techniques*, April 2008, pp.545-562.

[22] Yang J F, Cui A, Stolfo S, Sethumadhavan S. Concurrency attacks. In *Proc. the 4th USENIX Conf. Hot Topics in Parallelism*, June 2012.

[23] Mulliner C, Michéle B. Read it twice! A mass-storage-based TOCTTOU attack. In *Proc. the 6th USENIX Conf. Offensive Technologies*, August 2012, pp.105-112.

Kai Lu received his B.S. degree and Ph.D. degree in 1995 and 1999, respectively, both in computer science and technology, from the College of Computer, National University of Defense Technology, Changsha. He is now a professor in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems, parallel computing, and security.

Peng-Fei Wang received his B.S. and M.S. degrees in computer science and technology, in 2011 and 2013, respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now pursuing his Ph.D. degree in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems and software testing.

Gen Li received his B.S. degree and Ph.D. degree in computer science and technology, in 2004 and 2010, respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now an assistant professor in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems and software testing.

Xu Zhou received his B.S., M.S., and Ph.D. degrees in computer science and technology, in 2007, 2009, and 2014, respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now an assistant professor in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems and parallel computing.