What if Hardware is Untrusted?

A Study of the Hardware Double Fetch Problem in Linux Kernel

Kai Lu, Pengfei Wang, Gen Li, Xu Zhou
National University of Defense Technology
pfwang@nudt.edu.cn, wpengfeinudt@gmail.com

Abstract

The double fetch problem occurs when the data is maliciously changed between two kernel reads of supposedly the same data, which can cause serious security-related 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 in the I/O memory between the kernel and peripheral devices (aka. the **Hardware Double Fetch**). We proposed a static pattern-matching approach to identify the hardware double fetches from the Linux kernel. Our approach can analyze the entire kernel including all drivers without relying on the corresponding hardware. We identified 361 hardware double fetches, including 4 double-fetch vulnerabilities, which have been confirmed and fixed by the maintainers as a result of our report.

Introduction

The hardware-based attacks in recent years have placed attention on peripheral devices. The USB autoplay attack enables attackers to run executables by simply plugging a compromised USB device in the computer. The BadUSB from Blackhat 2014 performed attacks by writing malicious code onto USB control chips (i.e. firmware), which brought the risks from **what the devices carry** to the core of **how they work**. The underlying issue is the inability to guarantee and verify the functionality and integrity of the connected devices. With such a compromised hardware, attackers could completely take over a PC, invisibly alter files installed from the memory stick, or even redirect the users internet traffic, all without being detected.

"**Double fetch**" was named by Fermin J. Serna. It occurs *when the data is unexpectedly changed between two kernel reads of supposedly the same data*, which can cause security problems such as buffer overflows, information leakage, and system crashes. Previous research focused on the double fetches between the kernel and user applications, 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 peripheral devices, aka. the **Hardware Double Fetch**. In this study, we present the first dedicated work on this problem, which provides a new perspective to the double fetch problem by increasing the scope to include peripheral devices.

Motivation

Operating systems control peripheral devices by reading from and writing to the device registers via memory-mapped I/O. Due to the **absent of effective validation of the attached hardware**, compromised hardware could flip the data between two reads of the "same" I/O memory data.

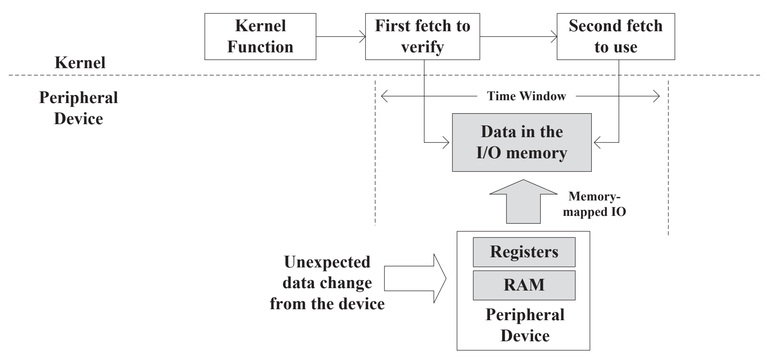

Figure 1: Illustration of How A Hardware Double Fetch Happens

As Figure 1 shows, a hardware double fetch problem occurs when the kernel reads the "same" I/O memory data twice, the first time to verify it and the second time to use it, assuming the data is unchanged. However, since the driver is unable to fully validate the attached hardware, compromised hardware could perform malicious data change between the two reads, causing data inconsistency for the kernel functioning. When the tampered data is related to memory accessing (e.g. message length indicators or queue pointers), security problem can be caused.

The hardware double fetch problem is **novel**, because the **involvement of hardware** makes a hardware double fetch different from the traditional double fetch in the following aspects:

- **Across different system boundaries.** A traditional double fetch occurs between the kernel and user applications, while a hardware double fetch occurs between the kernel and peripheral devices, which makes it a novel problem that no dedicated research was conducted on before.
- **Prior approaches are not workable.** Dynamic analysis adopted by prior traditional double fetch research relies on corresponding hardware to execute the program, and 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.
- **I/O memory has different sources.** Both device registers (including status registers, configure registers, and data registers) and device memory are mapped to I/O memory with consecutive addresses. Different data source has different functionality and thus different features (e.g. limited register bitwidth causes Linear Reads[1]).
- **Data change in the I/O memory.** The peripheral data 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.

Static Pattern-matching Approach

We proposed a static pattern-matching approach to identify the hardware double fetches in the Linux kernel. As is shown in Figure 2, the whole procedure of our approach is divided into four stages.

- **Identify:** Identify candidates based on the **basic pattern** of double fetch, i.e. *the consecutive invocations of wrapper functions*[2] *reading from the same I/O memory address in the same context.*
- **Switch:** Adopt a unified method by using one function to represent other similar wrapper functions, which **dramatically reduces the combinatorial situations** we need to match.
- **Refine:** Improve the precision by take into consideration of factors such as **interleaved write**, **pointer aliasing**, and **potential pointer change**.
- **Prune:** Remove cases that cannot cause double-fetch bugs, such as **linear reads**, **not double-used**, and **unused reads**.

[1] A value is transmitted separately, though reading from the same I/O address, everytime gets a different section of it.
[2] Dedicated functions provided by the Linux kernel to access the I/O memory, which is the only way to get I/O memory data

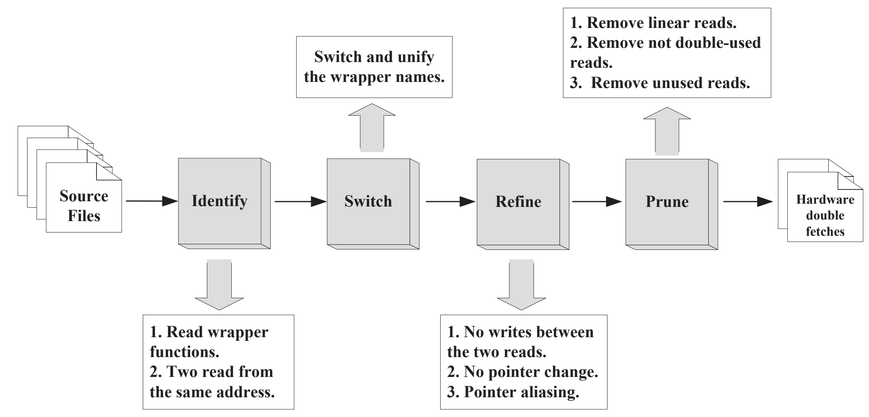

Figure 2: Overview of the Static Pattern-Matching Approach

Our approach is implemented based on the **Coccinelle matching engine**. The implementation is about 2.3 KLOC. We have made it publicly available online (https://github.com/wpengfei/hardware_df), hoping it can be useful for future study.

Results & Findings

Table 1: Identified Hardware Double Fetches Results

Types	Categories	Occurrences &Percentage	True Bugs
Status Regs	Common Check	59 (16.3%)	0
	Loop Check	80 (22.2%)	0
	Wait Check	81 (22.4%)	0
	Stable Check	18 (5.0%)	0
Configure Regs	Configure Check	29 (8.0%)	0
Data Regs	Check and Use	68 (18.8%)	3
Device 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 applied our approach to Linux kernel-4.10.1, which was the newest version when the experiment was conducted. The automatic pattern-matching process took approximately **28 minutes** and we got **361 occurrences** of hardware double fetches (Table 1) from **178 candidate files** out of 42,417 source files (.c or .h files) of the entire Linux kernel.

Then we manually reviewed these files to categorize the patterns and identify buggy cases. We analyzed each category with case studies to discuss the possibility of causing bugs. We reported the identified buggy cases to the kernel maintainers who made the final confirmation.

Based on the investigation of the results, we have the following findings:

- Hardware double fetches truly exist in the I/O memory. The occurrences range from all three types of device registers to the device memory.
- Hardware double fetches can cause bugs or even vulnerabilities (we identifed **4 double-fetch vulnerabilities** (listed in Table 2), which have never been found in the wild before.
- **Status registers** committed most of the identified hardware double fetches (65.9%, 238/361), but **data registers** are more likely to cause buggy situations (we found three) as they usually hold critical data related to memory accessing, such as message length variables and queue pointers.
- **Configure registers** are relatively safe as the kernel directs such registers rather than relying on them, thus the corrupted data from device configure registers can hardly harm the kernel.
- **Device memory** introduces the least hardware double fetches (only one) because double-fetching large blocks of data is usually avoided by developers for the concern of efficiency.

Table 2: Description of 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 message sequence-number value.
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.
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.
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.

Conclusions

- Presented **the first dedicated study to the double fetch problem in the I/O memory**, which provides a new perspective to the double fetch problem by increasing the scope to include peripheral devices.
- Proposed a static pattern-matching approach to identify hardware double fetches, which could **analyze all the driver code in one execution without relying on the corresponding hardware**.
- Identified 361 hardware double fetches from Linux kernel-4.10.1, including **4 previously unknown double-fetch vulnerabilities**. We provided patches, and all the vulnerabilities have been confirmed and fixed by the maintainers as a result of our report.
- Conducted a thorough investigation of the identified cases. The cases were categorized and analyzed with examples to discuss the possibility of causing bugs. We summarized findings based on our investigation.