CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2017; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/4345

A Survey of The Double-Fetch Vulnerabilities

Pengfei WANG[2*], Kai LU [123], Gen LI [2], Xu ZHOU[23]

[1]*Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha 410073, P.R.China*
[2]*College of Computer, National University of Defense Technology, Changsha 410073, P.R.China*
[3]*Collaborative Innovation Center of High-Performance Computing, National University of Defense Technology, Changsha 410073, P.R. China*

SUMMARY

Race conditions widely exist in concurrent programs, and concurrency errors caused by harmful races could lead to severe system failures. A double fetch is a typical situation when the system kernel inevitably accesses user space data multiple times, and it turns into a vulnerability when the data consistency is violated under a special race condition between kernel and user space. In this survey, we present the first (to the best of our knowledge) comprehensive study on double-fetch vulnerabilities in the real world. Our study is based on the investigation of 91 real-world double-fetch vulnerabilities collected from the CVE database and other relevant reports, which covers a period of recent 12 years. Our work reveals some interesting findings on the double-fetch vulnerabilities, ranging from the various occurrences across different kernels and system levels to the involvement of specific patterns. We also divide the consequences that are usually caused by the double-fetch vulnerabilities into four categories and discuss each, summarize viable exploitation techniques from existing works, provide useful guidances to detect and practical strategies to prevent double-fetch vulnerabilities.

Received: Jun. 9th 2017, Accepted: Sept. 6th 2017.

KEY WORDS: Double-Fetch Vulnerability; Race Condition between Kernel and User Space; Double-Fetch Exploitation; CVE Database Survey

1. INTRODUCTION

The widely use of multi-core hardware is making concurrent programs increasingly pervasive, especially in operating systems, real-time systems, and computing intensive systems. Concurrency not only improves processing efficiency but also facilitates program design. However, concurrent programs are also notorious for the difficulties in detecting hidden concurrency errors during in-house testing, which are introduced by the non-deterministic interleaving of the thread executions.

Data races are race conditions occur at memory access level, which are the most common causes of the concurrency errors. A data race occurs when two threads are accessing the same memory location, at least one of the two accesses is a write, and the relative ordering of the two accesses is not enforced by any synchronization primitives [1]. Data races could lead to hard-to-reproduce errors that are time-consuming to find and fix. They widely exist in concurrent programs, such as industrial

*Correspondence to: College of Computer, National University of Defense Technology, Changsha 410073, P.R.China. E-mail: pfwang@nudt.edu.cn
This is the peer reviewed version of the following article: P.Wang et al. (2017) *A Survey of The Double-Fetch Vulnerabilities*, which has been published in final form at https://doi.org/10.1002/cpe.4345. This article may be used for non-commercial purposes in accordance withWiley Terms and Conditions for Use of Self-ArchivedVersions.

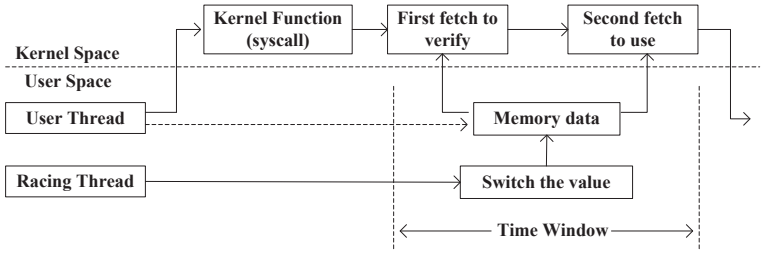

Figure 1. Description of A Double-fetch Vulnerability

control software, which can cause severe failures and real-world disasters, such as the Therac-25 accident[2], the jeopardy of Mars Pathfinder [3], the 2003 North American blackout[4], and the 2012 Facebook IPO delay incident[5], all of them resulted in significant casualties or property loss in real world.

In addition to the regular cases that occur among threads, races also exist between the kernel and user space, which is a special situation that very few work has been conducted on before. In 2008, Fermin J. Serna [6] firstly introduced the term "double fetch" to describe this situation. A double fetch occurs when the kernel (e.g. via a syscall) reads the same value that resides in the user space twice, the first time to verify the value or establish the relationship with the kernel while the second time to use the value [7]. As Figure 1 shows, a double fetch turns into a double-fetch vulnerability when there is a special race condition that occurs between kernel and user space, under which a concurrently running user thread changes the value within the time window between the two kernel reads. Then, when the kernel function fetches the value a second time to use, it gets a different one. The data inconsistency introduced by the double-fetch vulnerabilities could lead to problematic consequences such as privilege escalation, information leakage, kernel crash, etc. The famous KHOBE attack [8] is a typical application of the double-fetch vulnerability to bypass the security software on the system. Jurczyk and Coldwind [7] were the first to detect and exploit double-fetch vulnerabilities in their Bochspwn project. They detect double-fetch vulnerabilities based on memory access tracing and discovered a series of double-fetch vulnerabilities in the Windows kernel. They also suggested that there are still likely tens of such vulnerabilities in the kernel, especially the IOCTL handlers and third-party drivers.

In order to get a thoroughly understanding of the double-fetch vulnerability so as to prevent it, we conduct this comprehensive study. Our study is based on the investigation of 91 real-world double-fetch vulnerabilities collected from the CVE (Common Vulnerabilities and Exposures) database and other reports and patches from relevant repositories and archives. These vulnerabilities' occurrences cover a period of recent 12 years (date from 2005). By carefully examining each vulnerability case, we can gain deep insight into questions such as "How a double-fetch vulnerability happens?", "In what situations is a double-fetch vulnerability prone to happen?", "What consequences could a double-fetch vulnerability cause?", "How to exploit a double-fetch vulnerability?", "How to defend against the double-fetch vulnerabilities?" and in general gain a new level of insight into the common approaches and issues that underlie. To sum up, we make the following contributions:

- **First comprehensive study of the double-fetch vulnerabilities in real world**. Our work provides the first (to the best of our knowledge) comprehensive study of the double-fetch vulnerabilities in the real world. We investigate 91 (as many as we can find) real-world double-fetch vulnerabilities collected from the CVE database and other relevant reports. For each case, we examine the reports, corresponding source code, related patches and programmers' discussion, all of which together provide us a relatively thorough understanding of the patterns, manifestation conditions, and other characteristics of the double-fetch vulnerabilities.
- **Diversity of double-fetch vulnerabilities**. We demonstrated the diversity by providing and analyzing various cases, which led to interesting findings: Double-fetch vulnerabilities are fundamental problems that are independent of any program languages or operating systems.

DOI: 10.1002/4345

Their occurrences range from source code to pre-processed binaries, and even the peripheral devices can cause double-fetch vulnerabilities in the I/O memory. Double-fetch vulnerabilities involve certain patterns, such as ”size checking” and ”shallow copy”, etc.

- **Publicly available double-fetch vulnerability dataset**. Beyond these results, another outcome of our work is a local dataset of the double-fetch vulnerabilities, which contains all the cases and information of our work. We make it publicly available (https://github.com/wpengfei/df_dataset.git) for further study by kernel developers and security researchers.
- **Strategies on exploitation, detection and prevention**. We concluded four categories of consequences that the double-fetch vulnerabilities usually cause. We summarized viable exploitation techniques from existing works. We provided guidance on detection in two categories of dynamic and static approaches, which include how these approaches work, the advantages and disadvantages, and suggestions in implementation. We also gave prevention suggestions based on the anlysis of real-world double-fetch vulnerability patches we collected.

The rest of the paper is organized as follows: Section 2 discusses the related works of this topic. Section 3 presents the motivation of this work and the methodology we adopt. Section 4 introduces various situations in which a double-fetch vulnerability could happen, which reveals some interesting findings. Section 5 discusses the categorized consequences, the exploitation techniques, and implications of detection as well as strategies for prevention, followed by conclusions.

2. RELATED WORKS

So far, only limited research has been conducted on the research of double-fetch vulnerabilities, which includes Bochspwn [7] and the work of Wilhelm [9]. There are also some works that closely related to double-fetch vulnerabilities, such as the TOCTOU and KHOBE attack.

Jurczyk and Coldwind [7] carried out the first study on double-fetch vulnerabilities in their Bochspwn project. They instrumented the Windows kernel to identify double-fetch vulnerabilities dynamically by observing read operations on the same user space address within a very short time. They found real double-fetch vulnerabilities from Windows and proposed exploitation techniques. However, their analysis and findings were limited to Windows and their dynamic approach was limited to the code coverage. In our work, we give a more thorough study on various double-fetch vulnerability examples, including cases from preprocessed code and case from I/O memory. Besides, we also argued that static approaches are necessary under certain circumstances, such as for driver code when the hardware is unavailable for dynamic approaches, which they neglected.

Wilhelm [9] used an approach similar to the Bochspwn project to analyze memory access pattern of para-virtualized devices' backend components. His analysis discovered three novel security vulnerabilities in security critical backend components. Interestingly, one of the discovered vulnerabilities does not exist is the source code but is introduced through compiler optimization (see the discussion in Section 4.4.2) because the compiler optimizes the code in a way that a second fetch is conducted instead of reusing the value from the first fetch. However, although Wilhelm's work enriches the diversity of double-fetch vulnerability analysis, as a dynamic approach based detection work, it still lacks the thorough analysis of the double-fetch vulnerabilities, such as the specific patterns, the special case in I/O memory and the exploitation techniques. We will give a more comprehensive study on double-fetch vulnerabilities by an investigation of various real-world cases as well as an introduction to exploitation techniques and detection strategies.

A Time-Of-Check to Time-Of-Use (TOCTOU in short) race condition happens when a program checks for a particular characteristic of an object, so as to take some action based on the assumption that the characteristic still holds, whereas it actually does not hold any longer [10]. The data inconsistency in TOCTOU is usually caused by a race condition, which results from improper synchronized concurrent accesses to a shared object. There are varieties of shared objects in the computer system, such as files, sockets [11] and memory locations [12]. TOCTOUs are most

DOI: 10.1002/4345

common in Unix file system, which can date back to the 1990s. Numerous approaches [13] [14] [15] [16] have been proposed to solve this problem, but there is still no general, portable, secure way to solve it in the file system[17] because these approaches have to take into consideration of many specific aspects of this issue, such as the mapping between filenames and inode, the resolution of symbolic links, the UNIX file-system interface and implementation, etc. A double-fetch vulnerability is different from a typical TOCTOU issue that are specific to the shared objects (e.g. a file or a socket), because a double-fetch vulnerability focuses on the interaction between kernel and user space at memory access level, which is more likely to cause security related problems, and previous TOCTOU studies that specified for file system or other shared objects are not applicable anymore for the double-fetch issue. Our work gives a comprehensive study on double-fetch vulnerabilities, which includes the causes and consequences of a double-fetch vulnerability, as well as the approaches of double-fetch vulnerability exploitation and detection.

Yang et al [12] concentrated on concurrency errors under TOCTOU situations, and they believed such kind of bugs could be exploited to carry out concurrency attacks. They also studied some real cases to catalog concurrency attacks and pointed out that the risk of concurrency attacks is proportional to the duration of the vulnerability window. Their work is very similar to the double-fetch vulnerability we study except they consider user applications while a double-fetch vulnerability involves the kernel, which could cause a more serious impact. Their work is limited to a few user application analysis without practical description. In our work, we give a more comprehensive study with much more cases as well as analysis of causes, consequences, exploitation and detection strategies.

A double-fetch vulnerability in nature is caused by a race condition at memory access level. Plenty of works has been conducted on race conditions at memory access level (data races). Static approaches analyze the program without running it [18] [19] [20] [21] [22] [23] [24]. They can find the corner cases because of the better coverage of the code and the overall knowledge of the program. The major disadvantage is the false reports generated due to the lack of the program's full runtime execution context. Dynamic approaches actually execute the program to verify the races [25] [26] [27], checking whether a race could cause program failure in executions. They control active thread scheduler to trigger specific interleaving to increase the probability of manifestation [28, 24]. Runtime overhead is a severe problem, and testing of the driver code requires the support of specific hardware. However, A double-fetch vulnerability is caused by a special race condition between kernel and user space, and most of the previous race detection works did not take into consideration of this situation. Therefore, we give the first comprehensive study that specific to the characteristics of double-fetch vulnerabilities. In our work, we give a thorough analysis of double-fetch vulnerabilities based on various of real-world cases, including the causes, consequences, and specific patterns. We also provide exploitation and detection strategies that are specific to double-fetch vulnerabilities.

3. MOTIVATION

3.1. Racing Between Kernel and User Space

In the modern computer system, memory is divided into kernel space and user space. The kernel space is where the kernel code is stored and executes under, whilst the user space is a set of locations where normal user processes run. Since the kernel address space and the user address space are both virtual and physical isolated, special schemes are provided by the operating system to exchange data between kernel and user space. In Windows, we can use device input and output control (IOCTL) method or shared memory object method to exchange data between kernel and user space [29], which are very similar to the shared memory approach among user spaces, and data is accessed by dereferencing a pointer. While in Linux, some specific functions are provided to undertake this task. For instance, Linux has such functions as `copy_from_user()`, `copy_to_user()`, `get_user()`, `put_user()` etc. Here we use term *copy functions* to represent the functions that exchange data between kernel and user space in Linux. Programmers are not allowed to manipulate

DOI: 10.1002/4345

user data by directly dereferencing user pointer except using copy functions because copy functions provide safe data transfer by handling complicated situations such as page fault. Thus, any data exchange between kernel and user space in Linux should involve invocations of the copy functions.

Since some memory data that resides in user space is accessible by both the kernel and the user process, a race condition is likely to happen between kernel and user space. When the kernel reads twice of some data from the same user memory location, a user thread could rewrite the data between the two reads under a race condition, which introduces data inconsistency for the kernel use, and therefore a double-fetch vulnerability happens. However, a double-fetch vulnerability is different from a regular data race because the race condition is separated by the kernel and user space. For a data race, the read and write operations exist in the same address space, therefore most of the previous approaches detect data races by identifying both the read and write operations that accessing the same memory location. However, things are different for a double-fetch vulnerability. The kernel space only contains two reads while the write resides in the user space. The write in the user space is usually potential, which means it does not essentially exist when we analyzing the program, but it is likely to be created by a malicious user to race with the kernel. Besides, a double-fetch vulnerability is also different from a regular data race in the way of accessing data. The involvement of the kernel makes the data exchange relies on special schemes(e.g. copy functions in Linux and IOCTL in Windows) instead of dereferencing the user pointer directly, which means previous data race detection approaches are not applicable anymore. Moreover, as a semantic issue, a double fetch is more complicated than a regular data race. A double-fetch vulnerability requires a fetch (the check) phase and a use phase (where the fetched data is used). Although the check can be located by matching the patterns of fetch operations, the use of the fetched data varies a lot. Therefore, a double-fetch vulnerability is different from a regular data race, and we shall pay attention to it due to its specialty, and severity.

3.2. Double-Fetch Vulnerabilities Cause the KHOBE Attack

The KHOBE (stands for Kernel HOok Bypassing Engine) attack or the ”argument-switch attack”, was discovered by security research group Matousec on May 2010 [8]. It is essentially a typical application of the double-fetch vulnerability targeting at bypassing the security software on Windows, which caused an earthquake for the Windows desktops. Generally, the code running on the host computer is scanned by the security software to find whether it is malware or the data is safe. In the KHOBE attack, the malware passes innocuous code to be scanned by the security software, then switch it to malicious code in place of the innocuous code right after the scan but before it is used, so as to bypass the security check and perform malicious activities, such as install any variety of malware or rootkit to take over the system.

More specifically, the KHOBE attack is effective against user mode and kernel mode hooks, especially the SSDT(System Service Descriptor Table) hooks, which are the most common kernel hooks in today’s security software. The SSDT is an internal dispatch table within Microsoft Windows, which contains information about the services that are used by the operating system for dispatching system calls. SSDTs belong to the kernel mode part of system call interface implementation, which stores addresses of routines that user mode code can invoke indirectly as a result of the special system call instruction [8]. Therefore, most modern security software modifies addresses stored in the tables to point to its own routines, called `hook functions` or `hook handlers`, which perform various checks to the calls from user mode applications before invoking the system calls. The hook handler blocks a call if it endangers the system, otherwise, it invokes the original system service with the same parameters it got from the user mode application. The KHOBE attack works by calling the system service with parameters that will certainly pass the checks of the services’ hook handlers. Then a faker thread tampers the contents of the parameters to malicious values that would never pass the checks of the hook handler. If the tampering operation occurs after the security checks are done but before the original service is called, the attack will successfully bypass the security software.

As we illustrated previously, a typical double-fetch vulnerability works by changing the contents of a user memory region referred to by a pointer. The pointer that passed to the kernel is

DOI: 10.1002/4345

unchangeable, while the content resides in the user space can be changed under race condition. The KHOBE attack is more difficult, which concentrates on tampering the kernel handles. Even though the value of the handle itself cannot be changed because it is copied to the kernel stack, the actual meaning of the handle value can be changed. The handle type can be regarded as a special type of pointer (handles refer to kernel objects like pointers do to memory regions), therefore the attack on handle arguments has much in common with tampering pointers [8].

The KHOBE attack technique poses a significant security risk, and plenty of KHOBE vulnerabilities were discovered from the prevalent security software at that time. One of the biggest threats is that a KHOBE attack could be paired with a zero-day attack to bypasses host security software and then do further damage. This general type of attack is fairly common with an initial exploit loading additional malware to fully take over the system [30].

3.3. Methodology

Our study is conducted based on the thorough investigation of 91 real-world double-fetch vulnerabilities (detailed information is available online from our dataset: https://github.com/wpengfei/df_dataset). We conducted the vulnerability collection work by the following steps.

- These vulnerabilities are initially collected from related works, such as Bochspwn [7] and Wilhelm's work [9], which includes the cases they used for demonstration as well as the cases they found and reported.
- We mainly take advantage of the CVE (Common Vulnerabilities and Exposures) [†] database to find vulnerabilities by manually searching the keywords such as "double fetch","kernel race condition","TOCTOU", etc.
- Then we manually filtered out the right vulnerabilities from the result and record the useful information, which includes the year of report, vulnerable kernel versions, buggy files, causes, and consequences, etc.
- We also tried to obtain the buggy source code and corresponding patches from the relevant repositories and archives, which includes Linux repository on Github[‡], Linux Kernel Mailing List Archive[§], Kernel Bugzilla[¶], Redhat Bugzilla[‖], etc.
- Then we further analyzed the buggy source code and the corresponding patches to conclude specific patterns and strategies on prevention.
- For the cases that source code is unavailable, such as Windows, we analyzed the Microsoft Security Bulletin[**] to get useful information.

After having finished the above work, we have constructed a local dataset of 91 real-world double-fetch vulnerabilities, which covers a period of recent 12 years. Based on the investigation of these vulnerabilities, we performed a further study, which includes the diversity of double-fetch vulnerabilities, the categories of consequences, the exploitation techniques, and the strategies for detection. We will discuss them in detail in the following sections.

4. DIVERSITY OF DOUBLE-FETCH VULNERABILITIES

Double-fetch vulnerabilities can take place in various situations. A double-fetch vulnerability is kernel-independent, more likely to be caused by specific patterns, and even possibly be introduced in the pre-processed binaries and in the peripheral data from I/O memory. In this section, we provide investigation on eight particular double-fetch vulnerabilities to prove the diversity.

[†] http://cve.mitre.org/
[‡] https://github.com/torvalds/linux
[§] https://lkml.org/
[¶] https://bugzilla.kernel.org
[‖] https://bugzilla.redhat.com
[**] https://technet.microsoft.com/en-us/library/security

DOI: 10.1002/4345

```
1 void win32k_entry_point(…) {
2    ...
3    my_struct = (PMY_STRUCT) lParam;
4    if (my_struct ->lpData) {
5       cbCapture = sizeof(MY_STRUCT) + my_struct->cbData; //1st fetch
6       ...
7       my_allocation = UserAllocPoolWithQuota(cbCapture, TAG_SMS_CAPTURE))
8       if ( my_allocation != NULL) {
9          RtlCopyMemory(my_allocation, my_struct->lpData, my_struct->cbData);  //2nd fetch
10      }
11   }
12   ...
13 }
```

Figure 2. A Double-Fetch Vulnerability in File `win32k.sys`

4.1. Kernel-independent

The occurrences of double-fetch vulnerabilities are not specific to a certain kernel. Therefore, it is a general problem that all the kernel developer should pay attention to.

4.1.1. Famous In Windows Kernel Term "double fetch" was firstly introduced to describe a double-fetch vulnerability (as is shown later) in Windows kernel. The first detection work on double-fetch vulnerabilities was conducted by Bochspwn on Windows [7]. The first application of double-fetch vulnerabilities in KHOBE attack was also conducted on Windows. Therefore, the Windows kernel is of vital importance for the double-fetch vulnerability study.

Here is a double-fetch vulnerability (CVE-2008-2252) in `win32k.sys` from Windows kernel, which had been patched in MS08-061[31]. In this case, an inadequate pool allocation and a later memory pool overflow can be caused, which might lead to the execution of arbitrary code in kernel mode. We can see from Figure 2, `my_struct->cbData` is fetched twice at line 5 and line 9 respectively. However, the checking at line 8, which aims to guarantee the successful pool allocation, is no longer effective anymore after the second fetch of `my_struct->cbData` (line 9), causing the potential inconsistency between the check (branch at line 8) and the use of the fetched value (line 9). Therefore, a concurrently running thread may have the chance to switch `my_struct->cbData` to a large value, which will result in pool overflow. Note here `cbData` is accessed by dereferencing a pointer rather than explicit copy across address spaces, this is because Windows kernel uses IOCTL method or shared memory object method to exchange data between kernel and user space, which are manipulated in a similar way to the shared memory approach that dereferences a pointer, however, they are internally different. Double-fetch vulnerabilities from Window kernel also include CVE-2013-1332, CVE-2013-3888, CVE-2013-3907, etc.

4.1.2. Even Earlier in Linux Kernel Double-fetch vulnerabilities occur not only in Windows kernel but also in Linux kernel, which was even earlier than when the double fetch was named. In Linux kernel, exchanging data between the kernel and user space depends on the copy functions (e.g. `copy_from_user()`, `get_user()`). Hence, the occurrence of a double-fetch vulnerability involves multiple invocations of copy functions, which is a pattern easy to match for detection.

Figure 3 shows a double-fetch vulnerability [32] in Linux kernel-2.6.9, which was reported as CVE-2005-2490. In file `compat.c`, when the user controlled content is copied to the kernel by `sendmsg()`, the same user data is accessed twice without sanity check at the second time. This can cause a kernel buffer overflow and therefore could lead to privilege escalation. Function `cmsghdr_from_user_compat_to_kern()` works in two steps: it first examines the parameters in the first loop (line 151) and copies the data in the second loop (line 184). However, only the first fetch (line 152) of `ucmlen` is examined (line 159), while the second fetch (line 185) is free from constraints, which may cause an overflow in the copy operation (line 194), or allow local users to execute arbitrary code by modifying the message in a concurrently running thread.

Similar vulnerabilities also include CVE-2016-6136, CVE-2006-0039, CVE-2006-0457, etc. Moreover, another double-fetch vulnerability in file `commctrl.c` of Linux kernel-4.5 (CVE-2016-6480) even existed for over 10 years.

DOI: 10.1002/4345

```
140 int cmsghdr_from_user_compat_to_kern(struct msghdr *kmsg, unsigned char
141                                         *stackbuf, int stackbuf_size)
142 {
143     struct compat_cmsghdr __user *ucmsg;
144     struct cmsghdr *kcmsg, *kcmsg_base;
        ...
149     kcmsg_base = kcmsg = (struct cmsghdr *)stackbuf;
150     ucmsg = CMSG_COMPAT_FIRSTHDR(kmsg);
151     while(ucmsg != NULL) {
152             if(get_user(ucmlen, &ucmsg->cmsg_len))
153                     return -EFAULT;
156             if(CMSG_COMPAT_ALIGN(ucmlen) <
157                     CMSG_COMPAT_ALIGN(sizeof(struct  compat_cmsghdr)))
158                             return -EINVAL;
159             if((unsigned long)(((char __user *)ucmsg – (char __user*) kmsg-> msg_control)
160              + ucmlen > kmsg->msg_controllen)
161                     return -EINVAL;
                ...
166             ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
167     }
        ...
183     ucmsg = CMSG_COMPAT_FIRSTHDR(kmsg);
184     while(ucmsg != NULL) {
185             __get_user(ucmlen, &ucmsg->cmsg_len);
186             tmp = ((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
187                     CMSG_ALIGN(sizeof(struct cmsghdr)));
188             kcmsg->cmsg_len = tmp;
                ...
193             if(copy_from_user(CMSG_DATA(kcmsg),CMSG_COMPAT_DATA (ucmsg),
194                     (ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))))
195                     goto out_free_efault;
                ...
201     }
        ...
212 }
```

Figure 3. A Double-Fetch Vulnerability in File `compat.c`

4.1.3. Between Guest OS and Hypervisor In addition to occurring in a single system like Windows and Linux, a double-fetch vulnerability can also occur across systems, which is between the guest OS and hypervisor. One example is CVE-2016-9381, which was found in the shared memory between the guest OS and hypervisor of Xen. A local administrative user on the guest system can trigger this double-fetch vulnerability by changing certain data on the shared ring to potentially execute arbitrary code with QEMU privileges. On systems that do not use a device model stub domain (or other techniques for deprivileging qemu), the local administrative user on the guest domain can obtain host system privileges. Unfortunately, detailed information about this vulnerability was not disclosed. A similar vulnerability (CVE-2015-8550) was found in the backend driver of Xen (we will discuss it in the next subsection).

4.2. Patterns Play A Significant Role

A double-fetch vulnerability usually matches some patterns that are specific to context conditions.

4.2.1. Size Checking. In Linux, devices are represented as files, which can be accessed from user space in exactly the same way as other types of files. A device driver is linked to its corresponding device file in the kernel space, and it exchanges data between the device file and user process through the copy functions. Most of the time, copying data between user and kernel space is straightforward and is done via a single invocation of the copy function. However, things get complicated when the data has a variable type or a variable length. Usually, such data can be separated into the data's *header* which is followed by the data's *body*. In the following, we consider such data to be *messages*, which are most common in drivers. For a variable length message, the header is used to identify the size of the complete message. It is copied to the kernel first to get the message size, check for validity, and allocate a local buffer of the necessary size. Then a second fetch follows to copy the whole message, which also includes the header, into the allocated buffer. In this situation, a double fetch is inevitable. If the size is retrieved from the header of the second fetch and used, the kernel

DOI: 10.1002/4345

```
131 static long ec_device_ioctl_xcmd(struct cros_ec_dev *ec, void __user *arg){
            ...
134         struct cros_ec_command u_cmd;
135         struct cros_ec_command *s_cmd;
136
137         if(copy_from_user(&u_cmd, arg, sizeof(u_cmd))){
138                     return -EFAULT;
139
140         s_cmd = kmalloc(sizeof(*s_cmd) + max(u_cmd.outsize, u_cmd.insize),
                                    GFP_KERNEL);
            ...
145         if (copy_from_user(s_cmd, arg, sizeof(*s_cmd) + u_cmd.outsize)) {
146                     ret = -EFAULT;
147                     goto exit;
148         }
            ...
151         ret = cros_ec_cmd_xfer(ec->ec_dev, s_cmd);
            ...
161 }
```

```
187 static int cros_ec_cmd_xfer_i2c(struct cros_ec_device *ec_dev,
188                                 struct cros_ec_command *msg){
            ...
209         packet_len = msg->insize + 3;
210         in_buf = kzalloc(packet_len, GFP_KERNEL);
            ...
220         packet_len = msg->outsize + 4;
221         out_buf = kzalloc(packet_len, GFP_KERNEL);
            ...
229         out_buf[2] = msg->outsize;
            ...
231         /* copy message payload and compute checksum */
232         sum = out_buf[0] + out_buf[1] + out_buf[2];
233         for (i = 0; i < msg->outsize; i++) {
234                     out_buf[3 + i] = msg->data[i];
235                     sum += out_buf[3 + i];
236         }
237         out_buf[3 + msg->outsize] = sum;
            ...
286 }
```

Figure 4. A Double-Fetch Vulnerability in File `cros_ec_dev.c`

becomes vulnerable as a malicious user could have changed the size element of the header between the two fetches. If for example, the size is used to control buffer access, this situation turns into a vulnerability [33].

Figure 4 shows a double-fetch vulnerability (CVE-2016-6156) in file `cros_ec_dev.c` from the driver of Linux kernel-4.5 [34], which is also a case that involves variable length message. Function `ec_device_ioctl_xcmd()` fetches data from user space pointed by pointer `arg` via `copy_from_user()` twice in line 137 and line 145 respectively. The first fetched value (`u_cmd`) is used to calculate a buffer size and to allocate a buffer of the calculated size (line 140), while the second copy fetches the whole message to the buffer (`s_cmd`) based on the calculated size. Therefore, the second fetch is based on `outsize` and `insize` from the first fetch. However, after the second fetch, the whole message is passed as a parameter to function `cros_ec_cmd_xfer` (line 151), in which the `outsize` and `insize` are used again (line 209, 220, 229, 233, 237). Since `outsize` and `insize` which now come from the second fetch are potentially inconsistent with the ones from the first fetch, and `outsize` also controls a loop (line 233), therefore, a double-fetch vulnerability occurs because malicious change of `outsize` or `insize` from concurrently running thread under race condition would cause kernel out-of-boundary access. Similar vulnerabilities also include CVE-2015-1420, CVE-2016-5728, CVE-2016-6130, CVE-2016-6136, CVE-2016-6480.

4.2.2. Shallow Copy As we illustrated previously, the data exchanged between the kernel (especially the drivers) and user space is usually in the form of a `message`. A message not only has variable length and type but also has variable structure members, in which pointers are common to see. Exchanging of messages that have pointer members in the structure would lead to a situation we call as `shallow copy`, which could also cause double-fetch vulnerabilities. A shallow copy happens when a data struct (the first buffer) in the user space is copied to the kernel space and the struct contains a pointer to another region in user space (the second buffer). The copy function only copies the first buffer (a shallow copy) and the second buffer has to be copied explicitly by a second invocation of the copy function (to perform a deep copy). It is common to see that user processes copy data from user space into kernel space, and copy the data back after processing it. Such data is usually in a second buffer in user space (a pointer member of a message) and pointed to by a pointer in the first buffer (a pointer to the message). Deep copies of such data will cause multiple invocations of copy functions, which causes a double fetch which is very likely to turn into a double-fetch vulnerability.

In Linux, performing a deep copy on a message that has a second buffer can cause a double-fetch vulnerability. Figure 5 shows such a case (CVE-2016-6130) in file `sclp_ctl.c` from Linux kernel-4.5 [35]. Function `sclp_ctl_ioctl_sccb()` performs a shallow copy of a data structure from user space pointed to by `user_area` into `ctl_sccb` (line 61). To do a deep copy it has to copy another data structure (the second buffer) from user space pointed to by `ctl_sccb.sccb` (line 74). However, the size of the data structure is variable, in order to copy the

DOI: 10.1002/4345

```
55 static int sclp_ctl_ioctl_sccb(void __user *user_area)
56 {
57          struct sclp_ctl_sccb ctl_sccb;
58          struct sccb_header *sccb;
            ...
61          if (copy_from_user(&ctl_sccb, user_area, sizeof(ctl_sccb)))
62                  return -EFAULT;
            ...
65          sccb = (void *) get_zeroed_page(GFP_KERNEL | GFP_DMA);
66          if (!sccb)
67                  return -ENOMEM;
68          if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb), sizeof(*sccb))) {
69                  rc = -EFAULT;
70                  goto out_free;
71          }
            ...
74          if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb), sccb->length)) {
75                  rc = -EFAULT;
76                  goto out_free;
77          }
            ...
78          if (copy_to_user(u64_to_uptr(ctl_sccb.sccb), sccb, sccb->length))
79                  rc = -EFAULT;
            ...
86 }
```

Figure 5. A Double-Fetch Vulnerability in File `sclp_ctl.c`

```
571 static long ioctl_file_dedupe_range(struct file *file, void __user *arg)
572 {
573         struct file_dedupe_range __user *argp = arg;
574         struct file_dedupe_range *same = NULL;
            ...
579         if (get_user(count, &argp->dest_count)) {
580                 ret = -EFAULT;
581                 goto out;
582         }
583
584         size = offsetof(struct file_dedupe_range __user, info[count]);
585
586         same = memdup_user(argp, size);
            ...
593         ret = vfs_dedupe_file_range(file, same);
594         if (ret)
595                 goto out;
            ...
604 }
```

```
1569 int vfs_dedupe_file_range(struct file *file,
                    struct file_dedupe_range *same){
         ...
1578     u16 count = same->dest_count;
         ...
1606     for (i = 0; i < count; i++) {
1607         same->info[i].bytes_deduped = 0ULL;
1608         same->info[i].status = FILE_DEDUPE_RANGE_SAME;
1609     }
1610
1611     for (i = 0, info = same->info; i < count; i++, info++) {
             ...
1656     }
1669 }
```

Figure 6. A Double-Fetch Vulnerability in File `fs/ioctl.c`

data, it first fetches the header of the data structure into the newly created kernel space pointed to by `sccb` (line 68) to get the data length in `sccb->length`. Then, based on `sccb->length`, it copies the whole content with a second fetch at line 74. Finally, at line 78, the data is copied back to the user space. While it looks like both invocations of the copy functions at lines 74 and 78 use the same length `sccb->length`, line 78 actually uses the value from the second fetch (line 74) while line 74 uses the value from the first fetch (line 68). Since a malicious user thread may have changed value `sccb->length` under race condition between the two fetches, a kernel information leakage could be caused at line 78 when copying back data to user space with a larger size. Therefore, this is a double-fetch vulnerability.

4.2.3. Inter-Procedure. In addition to the above situations that the two fetches of a double-fetch vulnerability take place in the same procedure, a double-fetch vulnerability can also be inter-procedure, which is more complicated and difficult to detect. Inter-procedure situations will cause false negatives for static-based approaches that match the two fetches to detect double-fetch vulnerabilities. Besides, the fetched value is passed as parameters to other procedures, which will cause difficulty in identifying how the value is used and how the problem is caused.

DOI: 10.1002/4345

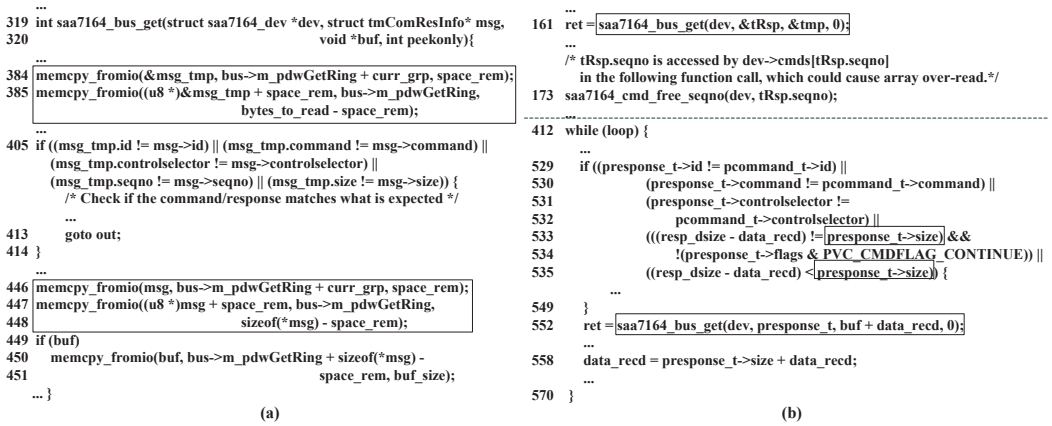

Figure 7. A Double-Fetch Vulnerability from Device I/O Memory

Figure 6 shows a double-fetch vulnerability (CVE-2016-6516) in file `fs/ioctl.c` from Linux kernel-4.7 [36], which is also a inter-procedure case. In function `ioctl_file_dedupe_range()`, user data `arge->dest_count` is fetched for the first time at line 579, and used to calculate `size` at line 584. Instead of a second data fetch, it duplicates a user memory region (line 586) of `size`. Then the duplicated memory region is passed as a parameter to another function `vfs_dedupe_file_range()`. In this function, `dest_count` is fetched again (line 1578) from the duplicated memory region, and used at line 1606 and line 1611 respectively, both of which are used as counters to control the loops. Since a malicious user might have changed the user data `dest_count` under race condition between the first fetch and the memory duplication, the counter retrieved from the duplicated memory might be larger that supposed to, and consequences such as heap-based buffer overflow or possibly privilege escalation would be caused by this double-fetch vulnerability.

4.3. Peripheral Devices Are Not Safe

A double-fetch vulnerability occurs not only in the memory between the kernel and user space, but also in the I/O memory between the kernel and peripheral device. As is known to all, every peripheral device is controlled via the writing and reading of its registers and device memory by the kernel, and when drivers use the memory-mapped I/O method [37, 38, 39, 40] to establish the I/O with the devices, reading from and writing to the I/O memory works in exactly the same way as accessing the regular memory in the computer. If the data from the same I/O memory address changes between two reads and the kernel then uses it as the same data without noticing, a double fetch problem occurs. In addition, since drivers are essentially unable to fully validate the attached devices, a compromised device can be controlled to perform malicious data change with the correct timing required to trigger this double-fetch vulnerability.

Figure 7 (a) is a code snippet from `/drivers/media/pci/saa7164/saa7164-bus.c` in linux-4.10.1, showing such a double-fetch vulnerability (CVE-2017-8813). Function `saa7164_bus_get()` uses line 384 and 385 to fetch I/O memory data from `bus->m_pdwGetRing` to local buffer `msg_tmp`, then it uses `msg_tmp` to do a series of checks with the desired data at line 405, including `->size` and `->seqno`. After the checks, a second fetch (line 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` of the secondly fetched message changes, using that message is likely to cause memory access related problems.

Figure 7 (b) shows how this function is called. At line 552, function `saa7164_bus_get()` is called in a loop to receive data. The fetched message is stored in `presponse_t`, and `->size` field is used to calculate the position pointer for `buf`. However, the developer has aware of this problem

Prepared using cpeauth.cls
DOI: 10.1002/4345

```
...
669 #define hlist_entry_safe(ptr, type, member) \
670         (ptr) ? hlist_entry(ptr, type, member) : NULL
...
```

Figure 8. A Double-Fetch Vulnerability in a Macro

and added a validation check (from line 529 to 535) to prevent potential invalid message. Therefore, this situation is safe. Another situation occurs at line 161, the fetched message is stored in `tRsp` and is immediately passed to function `saa7164_cmd_free_seqno()` (line 173), within which `->seqno` field of the message is used to access an array (`dev->cmd[tRsp.seqno]`). But this time, the message is used without an additional validation, if `->seqno` field of the message is changed to a very large value before the second fetch, an array over-access error could be caused. Therefore, causing a double-fetch vulnerability.

4.4. Beyound Ordinary Source Code

4.4.1. Macro Substitution In addition to occurring in the ordinary source code, a double-fetch vulnerability could also occur at a lower level, such as in the macro before pre-processing. Macros are different from the ordinary source code because macros expand (or substitute) during pre-processing. In other words, there could be multiple occurrences of macro substitutions in the source code, sometimes even the programmer cannot figure out where exactly a macro expands. For the ordinary source code, we could be aware of the double-fetch problem when copying the data from user space to kernel space because a double-fetch problem happens exactly where the code is. However, things are different for a macro. Most of the time, it is difficult for a programmer to take into consideration of all the substitutions when writing a macro, let alone the substitution contexts. Therefore, for some substitutions, a double-fetch situation could lead to a vulnerability, while for others, it will not. In short, it is hard for a macro to predict the substitution contexts, thus unexpected consequences can be caused, such as a double-fetch vulnerability.

Figure 8 shows a double-fetch vulnerability in the macro [41], which was in file `include/linux/list.h` of Linux kernel-3.9-rc1 and had been patched in rc-3. This macro fetches the pointer `ptr` twice, the first time to test for NULL and the second time to compute the offset back to the enclosing structure (line 670). However, due to the potential pointer change between the two fetches, a null-pointer crash might be caused. This situation makes a double-fetch vulnerability particularly difficult to find because most of the approaches applied to source code only after expanding the macros, such as LLVM works on the AST (Abstract Syntax Tree). Very few approaches support the analysis of macro before expansion.

4.4.2. Compiler Optimization Finally, a double-fetch vulnerability can also be introduced into the binaries during compilation, which is even harder to notice than occurring in the macros. CVE-2015-8550 [42] is such a double-fetch vulnerability introduced by compiler optimization of Xen Hypervisor, which caused problems in the communication between the frontend driver and backend driver components. What makes this case particularly interesting is that this double-fetch vulnerability didn't show up when inspecting the source code, but it could clearly be seen in the compiled binary.

As is shown in Figure 9, the left part is the source code of a switch statement in xen-pciback (the backend component used for para-virtualized PCI devices), within which no sign of any double-fetch situation is found. While the right part of Figure 9 is the assembling of this switch statement, and we can observe a double-fetch situation occurs. During the compilation, the compiler generates a jump table to dynamically jump to the correct branch in the switch statement. The `r13` register points to a shared memory region and `r13+0x4` corresponds to the value used in the switch statement to select a branch. The value of `r13+0x4` is used to compare with the upper limit `0x5` (line 1). If it is higher than `0x5`, the default branch (line 2) is used. In line 4, `r13+0x4` is fetched a second time and used as an index of the jump table at line 5. As a result, if the data stored in

DOI: 10.1002/4345

```
switch(op->cmd){
  case XEN_PCI_OP_conf_read:
      op->err = xen_pcibk_config_read(dev,
                op->offset, op->size, &op->value);
      break;
  case XEN_PCI_OP_conf_write:
      //...
  case XEN_PCI_OP_enable_msi:
      //...
  case XEN_PCI_OP_disable_msi:
      //...
  default:
      op->err = XEN_PCI_ERR_not_implemented;
}
```

```
1  cmp  DWORD PTR [r13 + 0x4], 0x5
2  mov  DWORD PTR [rbp - 0x4c], eax
3  ja   0x3358 < xen_pcibk_do_op + 952 >
4  mov  eax, DWORD PTR [r13 + 0x4]
5  jmp  QWORD PTR [rax*8 + off_77D0]
```

Figure 9. A Double-Fetch Vulnerability in The Compiled Binary

`r13+0x4` is modified between the two fetches, the final jump (line 5) destination can be affected and arbitrary code can potentially be executed, which possibly lead to privilege escalation.

This double-fetch vulnerability occurs because if the pointers to the shared memory regions are not labeled as *volatile* during compiling, the compiler is allowed to turn a single memory access (in the code to be compiled) into multiple accesses at the binary level since it assumes that the memory will not be changed by another thread of execution. Note that the user data here looks like is fetched by dereferencing a pointer without using the copy function, however, this is a special case that has the same result. Although in Linux, programmers are not allowed to fetch user space data by dereferencing a pointer without using a copy function, internally, copy functions are implemented by assembling and data is actually transferred by dereferencing user pointers. Therefore, in the case of Figure 9, the assembling is more like how it works within a copy function. In other words, this special case is an assembling level double-fetch vulnerability, which is lower than ordinary ones.

This compilation introduced double-fetch vulnerability had been patched already, and even if the real world impact is probably quite low, this case enriches the diversity of double-fetch vulnerabilities.

5. DISCUSSION

Double-fetch vulnerabilities can cause serious consequences that are exploitable by the attackers. Existing works have proved the feasibility of exploiting double-fetch vulnerabilities. In this section, we categorize the consequences caused by double-fetch vulnerabilities, summarize practical approach on exploiting double-fetch vulnerabilities, and provide viable strategies on detection.

5.1. Double-fetch Vulnerabilities Cause Serious Consequences

According to our investigation of 91 double-fetch vulnerabilities, we categorized the consequences that are usually caused by the double-fetch vulnerabilities into the following four categories. These consequences are declared by either the CVE entries or the relevant reports. The percentage of involvement for each category is shown in Figure 10.

- **Gain privileges**. In most of the cases, gaining privilege is the ultimate goal for an exploitation work, which means taking full control of the whole system, such as CVE-2008-2252, CVE-2013-1248, etc. According to our investigation, there are totally 42 double-fetch vulnerabilities involve with this consequence, accounting for 46.2%.
- **Information leakage**. A double-fetch vulnerability can also make the user process illegally access additional kernel memory, and the user process could, therefore, obtain sensitive information from the kernel memory, leading to information leakage, such as CVE-2005-0457, CVE-2016-6130 etc. We found 35 such occurrences, accounting for 39.5%.
- **Bypassing**. Bypassing is another consequence that can be caused by a double-fetch vulnerability. The famous KHOBE attack [8] is a scenario of using the double-fetch vulnerability to bypass the security software. In addition, an attacking can also bypass the

DOI: 10.1002/4345

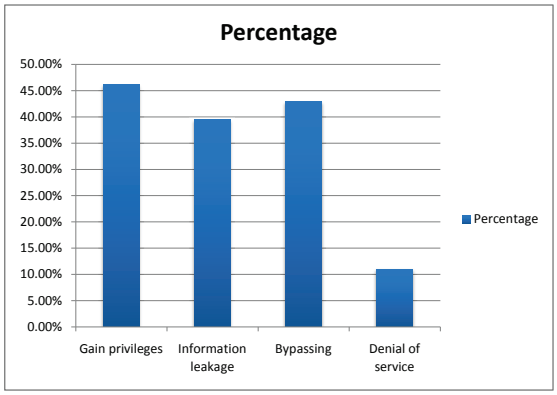

Figure 10. Comparison of Concequences Caused by Double-fetch Vulnerabilities

auditing system (CVE-2016-6236) or other system protection schemes. We observed 39 such occurrences (including the KHOBE attacks), accounting for 42.9%.

- **Denial of service**. Denial of service is a category of consequences that a software application or operating system stops functioning properly and exits, which may appear to hang until a crash reporting service reports the crash and any details relating to it [43]. A double-fetch vulnerability usually causes a denial of service in two ways: buffer over-read (or out-of-bounds array access) and buffer overflow. When the double-fetched value in a double-fetch vulnerability is a variable (e.g. indicating an array length of 128) that controls the accesses to an array of memory locations (usually in the form of a loop), the variable could be switched to a larger value (e.g. 256) right after the first fetch to check but before second fetch to use. Therefore, when the malicious value is used to control the memory access, if the access is a read, then buffer over-read is caused (e.g. CVE-2016-6156), if the access is a write, then buffer overflow is caused (e.g. CVE-2016-6516). Both of them can cause a denial of service. We found 10 occurrences of this category, accounting for 11.0%.

In addition, a double-fetch vulnerability could also cause combinatorial consequences. In our investigation, we found 30 cases (from CVE-2013-1248 to CVE-2013-1277) that can cause both information leakage and gain privileges, accounting for 33.0%. Three cases (CVE-2006-0457, CVE-2016-5728, and CVE-2015-1420) could cause information leakage and denial of service at the same time, accounting for 3.3%. And two cases (CVE-2016-6516 and CVE-2015-8550) could cause a denial of service and gain privileges, accounting for 2.2%.

5.2. Exploiting Double-fetch Vulnerabilities Is Indeed Viable

In 2012, Yang et al. [12] carried out a preliminary study on concurrency attacks based on the exploitation of Time-Of-Check to Time-Of-Use errors in concurrency programs. They proved the feasibility of such concurrency attack by real-world cases. They also pointed out that the exploitability of such a concurrency error depends on the duration of the timing window within which the error occur, and attackers can increase this window through carefully crafted inputs. These exploitable concurrency errors they studied are very similar to double-fetch vulnerabilities except they are in user applications while a double-fetch vulnerability involves the kernel.

In contrast to typical race condition errors where the program itself is the one creating the threads and running them, a double-fetch vulnerability exploitation requires the attacker himself to create the racing threads [44]. Jurczyk and Coldwind not only gave a systematic study of double-fetch vulnerabilities in Windows but also provided practical exploitation techniques. Tricks like using page boundary, disabling page cacheability, and TLB (Translation Lookaside Buffers) flushing could significantly expand the time window between the two kernel reads, so as to increase the success rate of changing the data [7].

DOI: 10.1002/4345

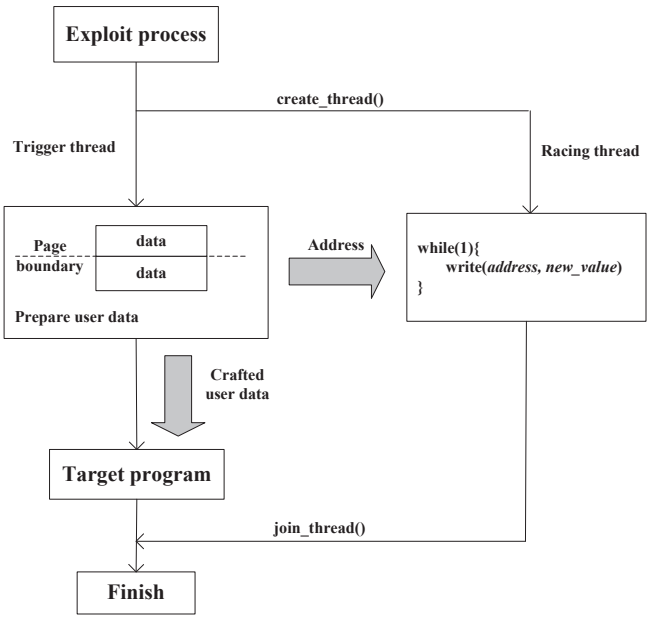

Figure 11. Illustration of Exploitating a Double-fetch Vulnerability

After the pilot work of Jurczyk and Coldwind, double-fetch vulnerabilities have raised awareness and plenty of works have been conducted on this, especially on the exploitation. Hammou [44] exploited a double-fetch vulnerability in a Windows driver, and escalated privileges on the system. The vulnerable kernel-mode driver (rcdriver) registers an IOCTL handler (RcIoCtl) for data copy between kernel and user space. The driver firstly accesses the IRP SystemBuffer to get the address (pointed by `UserAddress`) to where the user input resides, then performs `ProbeForWrite` on it to get the data. However, a double-fetch vulnerability happens because `UserAddress` is fetched twice from user mode memory. The exploitation works by supplying a valid user-mode address to `ProbeForWrite` and then quickly changes the `UserAddress` field to a kernel-mode address, so that to write 4 bytes data to any kernel memory location. The privilege escalates when the `TokenObject->Privileges.Enabled` field of an attacker's process is set to 0xffffffff in this way.

In addition to the above successful exploitations, there are some other attempts. Maiki [45] provided strategies on exploiting CVE-2005-2490. He pointed out that placing the crafted user data across page boundary could increase the time window. This is because in this way, it is more likely to trigger a page fault when accessing the data from two pages (or even more pages). A page fault could suspend the current thread that is fetching the user data, and wait for the desired page to be swapped in, while a malicious thread gets the chance to be scheduled (on a single-core machine) or is still running (on a multi-core machine) and switches the data before the fetching thread backs to work. Finally, the use of the changed data could cause a stack overflow and crash the kernel. Bauer [46] gave an exploitation PoC (Proof of Concept) of CVE-2016-6516 in a similar way, which could control which cache the overflow happens on and overwrites the fetched data with zeros.

Finally, as is shown in Figure 11, we summarized a common routine of exploiting a double-fetch vulnerability. The exploitation works by injecting two threads into the application, one that invokes the target program (trigger thread) and a second one that keeps switching the fetched data (racing thread). Threads running in the same address space prevents potential segmentation fault when rewriting user data. A race condition is triggered when the crafted user data is accessed concurrently by both the trigger thread to invoke the target program and the racing thread to rewrite the data. In order to increase the successful rate of changing data, tricks like placing data across page boundary, disabling page cacheability, and flushing TLB could be adopted to expand the time window between the two kernel reads.

DOI: 10.1002/4345

So far, to the best of our knowledge, there havent been any known real attacks based on the double-fetch vulnerabilities except the demos and PoCs on exploiting the already known ones proposed by the researchers. However, known vulnerabilities, especially the one-day vulnerabilities, are also dangerous since many systems do not apply the patches immediately. For example, CVE-2015-1426 is a double-fetch vulnerability in the Linux kernel. It was found in 2015 and then patched by the maintainer in the supported versions. However, in 2016, Wang et al [33] found it again in the latest Android kernel. This is because an Android kernel uses a long-time-support version of Linux as its kernel, which is relatively old. The patch for this vulnerability hadnt yet been pushed to the Android side from the Linux side, and thus caused a vulnerable Android kernel. Therefore, an attacker has two ways to find and exploit a system in terms of such vulnerability. One is to detect it from the systems by original methods, which is time-consuming but can discover zero-day vulnerabilities. Another way is to exploit known (one-day) vulnerabilities on the systems that havent applied the patch. We have added this part to the paper.

5.3. Implications on Detection

Detection of this vulnerability is one of the most important parts in the study of double-fetch vulnerabilities. In this section, we provide two viable approaches: dynamic memory access tracing and static pattern matching.

(1)**Dynamic memory access tracing**. A dynamic approach is easier to follow and construct, and both Bochspwn [7] and Wilhelm [9] adopted this one to detect double-fetch vulnerabilities. Based on a set of criteria that "at least two memory reads from the same virtual address", "both read operations take place within a short time frame (same semantic context)", "the code instantiating the reads must execute in kernel mode (ring-0)", "the virtual address subject to multiple reads must reside in memory writable by user threads (ring-3)", a dynamic approach by memory access tracing of the target OS running in a simulator (e.g. Bochs[††]) is proved effective and discovered a series of double-fetch vulnerabilities.

However, a dynamic approach is limited in the coverage it can achieve, and it also cannot be applied to code that needs corresponding hardware to be executed, for example, it cannot analyze all of the drivers without having the corresponding hardware or a simulation of it to run the test. Failing to do so will cause false negatives when using a dynamic approach to detect double-fetch vulnerabilities. For example, Bochspwn also tried to detect double-fetch vulnerabilities in Linux kernel of version 3.5 [47] but found nothing, while three vulnerabilities that were founded later actually existed in that version (CVE-2016-6136, CVE-2016-6480, CVE-2015-1420). The was because they were in the driver while Bochspwn failed to cover this part. Besides, runtime overhead is another problem, e.g. the simulator of Bochspwn needs 15 hours to boot up. Even though the dynamic approach is not perfect, sometimes it is the only choice when the kernel source code is unavailable to conduct a static analysis, such as the analysis of Windows kernel.

(2)**Static pattern matching**. A static approach is a good choice when the source code of the target kernel is available. Since a double-fetch vulnerability has obvious patterns that "occurs in the kernel source code", "two kernel reads of the user data via the same user pointer", "no kernel write to the user data between the two reads", a static pattern matching is effective and portable. Furthermore, Linux fetches data from user space by invocations of *copy functions*, which is also a feature that suits pattern matching. Besides, static-based approaches achieve better code coverage, and some of them support full path exploration. For example, Coccinelle [48] is a program matching and transformation engine that is used for finding and fixing bugs in systems code. It is path-sensitive but insensitive to newlines, spaces, comments, etc. Moreover, the matching of Coccinelle is applied without expanding copy functions that are defined as macros such as `get_user()` or `__get_user()`, which facilitates the identification of the kernel fetches by matching copy functions [33]. Besides, static approaches can also combine with a manual review of the source

[††]http://bochs.sourceforge.net/

DOI: 10.1002/4345

code, which could give a better understanding of how the vulnerability occurs or filter out the false reports.

However, one big shortcoming of the static approach is the false positives due to the lack of the overall knowledge of the runtime environment. In order to perform a precise analysis, the following problems should be addressed.

1. **Pointer aliasing**. Two fetches are not necessarily using the same pointer to fetch the data, but actually read from the same memory location. Neglecting this would lead to false negatives.
2. **Pointer changing**. Even using the same pointer, the pointer might be changed between the two fetches by adding an offset or assigned by another value, which results in pointing to a different memory location, therefore different data is fetched. Neglecting this could lead to false positives.
3. **Overlapped two fetches**. Two fetches might use two different pointers to read two different user data, but the two fetched data overlaps, which results in a double-fetched data section. For example, a message header vs the whole message. Neglecting this situation would cause false negatives.
4. **Inter-procedure analysis**. Two fetches can also respectively reside in two different procedures where one invokes another, and the use of the fetched data can even occur in a third procedure. In this situation, a simple pattern matching might be infeasible, and a call graph analysis should be helpful. Neglecting this situation might result in false negatives.
5. **Use of the double-fetched data**. The above-mentioned pattern only matches the operations where the user data is double-fetched. However, for a double-fetch vulnerability, the most important part is how the double-fetch data is used. If the double-fetched data is never used, then it will not cause a vulnerability.

In addition, hardware-based methods are proved to be effective in the detection of concurrency bugs. For example, Transactional Memory (TM) was once used to detect and prevent atomicity-violation concurrency bugs [49]. TM enables programmers to declare regions of code atomic without specifying a lock and has the potential to avoid these bugs. Studies [50, 51] found that TM is measurably easier than fine-grained locks, and results in fewer bugs and less development time, or better performance. However, these attempts were preliminary. They focused on using just a single mechanism (locks or transactions), while real-world large programs (such as kernel programs) use many different synchronization mechanisms, such as condition variables and I/O. Therefore, the TM-based approaches are not practical to detect double-fetch vulnerabilities in the kernel.

There are also proposals for race detectors with hardware assists. Some [52, 53, 54, 55] detect races by tagging the state in the caches as it is being accessed, and then piggybacking the tags on cache coherence protocol messages between processors so that they can be compared. The hardware can easily detect an address and an instruction involved in a race on the fly. Then, roll back and re-execute the code section to reveal the other instructions involved in the same race. SigRace [56] relies on automatically encoding signatures in the addresses of the data the processor is accessing. Then the signatures are automatically passed to a hardware module that intersects them with those of other processors. If the intersection is not null, a data race may have occurred. Another work based on hardware is DataCollider [57], which is a lightweight tool that detects data races in kernel modules. It randomly samples a small percentage of memory accesses as candidates for data-race detection, and it uses breakpoint facilities already supported by hardware architectures to lower runtime overhead. However, since these proposals detect races by identifying multiple threads accessing the same shared data, while the race condition in a double-fetch vulnerability is crafted by the malicious user when triggering the vulnerability, which does not necessarily exist when examining the program. Therefore these race detectors are not workable for the detection of double-fetch vulnerabilities. Hardware-assisted detection of double-fetch vulnerabilities remains immature.

DOI: 10.1002/4345

5.4. Strategies on Prevention

Based on the analysis of the real-world double-fetch vulnerability patches we collected, we summarized the following practical suggestions on preventing double-fetch vulnerabilities. We believe these suggestions are helpful for the OS developers to prevent such vulnerabilities.

1. **Don't copy the header twice**. For the "size checking" situation, a double-fetch vulnerability can be completely avoided if the second fetch only copies the message body and not the complete message which copies the header a second time.This approach is useful and was adopted by the patch of CVE-2015-1420.
2. **Use the same value**. A double fetch turns into a double-fetch vulnerability when the kernel uses the data both from the first fetch and the second fetch because a (malicious) user can change the data between the two fetches. If the developers only use the data from one of the fetches, problems are avoided. However, this approach causes the so-called "benign double fetch" situations [33], in which the double-fetched data is not double-used and thus not causing a vulnerability for now. Nevertheless, it can easily turn into a double-fetch vulnerability when the code is updated by a maintainer who reused the double-fetched data without pay special attention to this situation, which is a major disadvantage of this approach. CVE-2016-5728 is a vulnerability resulted from the update of a benign double fetch.
3. **Overwrite data**. There are also situations that some data needs to be fetched and used twice, for example, the complete message is passed to a different function for processing. One way to resolve the situation is to overwrite the data from the second fetch with the data that has been fetched first, so as to stick to the data from the first fetch. This approach is widely seen adopted in FreeBSD code [33], such as in `sys/dev/aacraid/aacraid.c`.
4. **Compare data**. Another way to resolve a double-fetch vulnerability is to compare the data from the first fetch to the data of the second fetch. If the data is not the same, the operation can be safely aborted. This approach is widely used to patch the double-fetch vulnerabilities (e.g. (CVE-2016-6130, CVE-2016-6156, CVE-2016-6480) since it does not need to change very much of the source code. Besides, it has the advantages of both allowing detecting attacks by malicious users and protecting from situations in which the data is changed without malicious intent (e.g., by some bug in user space code) [33].
5. **Labeling Volatile** For the special case that a double-fetch vulnerability introduced during the compiler optimization (CVE-2015-8550), we could simply prevent it by labeling the volatile option. In this way, the compiler would not turn a single memory access into multiple accesses at the binary level.

So far there hasn't been any formal framework (to the best of our knowledge) to prevent or fix such double-fetch vulnerabilities except one attempt. Wang et al [33] proposed an automatic tool to patch the discovered double-fetch vulnerabilities that belong to the size-checking category with the compare data approach.

A double-fetch vulnerability is very similar to the atomicity-violation bugs in concurrency programs. Both of them share the same principle that two accesses to the supposedly same data are interleaved by a remote thread access that changes the data under race condition. For such race-based atomicity violations, a general lesson is to keep the atomicity of accesses to the supposedly same data, so as to prevent the potential modification to the shared data. Since such atomicity violations usually occur between threads within the same process, synchronization primitives, such as locks, are helpful in keeping the atomicity. Even though synchronization primitives are less helpful for the prevention of double-fetch vulnerabilities since it introduces severe overhead in the kernel, we provide above practical solutions that are in accordance with the double-fetch characteristics.

DOI: 10.1002/4345

6. CONCLUSIONS

Double-fetch vulnerabilities can cause serious consequences in the kernel. This work provides the first (to the best of our knowledge) comprehensive study on double-fetch vulnerabilities. Our study is based on the investigation of 91 real-world double-fetch vulnerabilities collected from the CVE database and other relevant reports and patches, which covers a period of recent 12 years. We constructed a local dataset and made it publicly available for further study. Our work reveals some interesting findings on the double-fetch vulnerabilities, which include the various occurrences across different kernels and system levels, the involvement of specific patterns (e.g. "size checking" and "shallow copy"), and occurrence of different system layer (e.g. preprocessed code, I/O memory .etc). We also categorized four popular consequences that are usually caused by double-fetch vulnerabilities, provided viable exploitation techniques from existing works, discussed guidances to detection, and provided useful strategies to prevent double-fetch vulnerabilities.

ACKNOWLEDGEMENTS

The authors would like to sincerely thank all the reviewers for your time and expertise on this paper. Your insightful comments help us improve this work. This work is partially supported by the National High-tech R&D Program of China (863 Program) under Grants 2015AA01A301, by the program for New Century Excellent Talents in University, by the National Science Foundation (NSF) China 61272142, 61402492, 61402486, 61379146, 61272483, 61472437, and by the laboratory pre-research fund (9140C810106150C81001).

REFERENCES

1. Wu Z, Lu K, Wang X, Zhou X, Chen C. Detecting harmful data races through parallel verification. *The Journal of Supercomputing* 2015; **71**(8):2922–2943.
2. Leveson NG, Turner CS. An investigation of the therac-25 accidents. *Computer* 1993; **26**(7):18–41.
3. Wikipedia. Mars pathfinder 1997. URL https://en.wikipedia.org/wiki/Mars_Pathfinder#cite_note-24.
4. Jesdanun A. General electric acknowledges northeastern blackout bug 20ww. URL http://www.securityfocus.com/news/8032.
5. Net X. Nasdaq ceo blames software design for delayed facebook trading 2012. URL http://www.cs.com.cn/english/com/201205/t20120521_3337908.html.
6. Serna FJ. 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/.
7. Jurczyk M, Coldwind G. Identifying and exploiting windows kernel race conditions via memory access patterns. *Technical Report*, Google Research 2013. http://research.google.com/pubs/archive/42189.pdf.
8. Matousec. Khobe 8.0 earthquake for windows desktop security software 2010. URL http://www.matousec.com/info/articles/khobe-8.0-earthquake-for-windows-desktop-security-software.php.
9. Wilhelm F. Tracing privileged memory accesses to discover software vulnerabilities. Master's Thesis, Karlsruher Institut für Technologie 2015.
10. Bishop M, Dilger M, *et al.*. Checking for race conditions in file accesses. *Computing systems* 1996; **2**(2):131–152.
11. Watson RN. Exploiting concurrency vulnerabilities in system call wrappers. *First USENIX Workshop on Offensive Technologies*, WOOT '07, 2007.
12. Yang J, Cui A, Stolfo S, Sethumadhavan S. Concurrency attacks. *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, USENIX Association, 2012.
13. Chen H, Wagner D. Mops: an infrastructure for examining security properties of software. *Proceedings of the 9th ACM conference on Computer and communications security*, ACM, 2002; 235–244.
14. Cowan C, Beattie S, Wright C, Kroah-Hartman G. Raceguard: Kernel protection from temporary file race vulnerabilities. *USENIX Security Symposium*, 2001; 165–176.
15. Lhee KS, Chapin SJ. Detection of file-based race conditions. *International Journal of Information Security* 2005; **4**(1-2):105–119.
16. Payer M, Gross TR. Protecting applications against TOCTTOU races by user-space caching of file metadata. *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, 2012.
17. Cai X, Gui Y, Johnson R. Exploiting unix file-system races via algorithmic complexity attacks. *30th IEEE Symposium on Security and Privacy*, 2009; 27–41.
18. Voung JW, Jhala R, Lerner S. Relay: static race detection on millions of lines of code. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ACM, 2007; 205–214.
19. Pratikakis P, Foster JS, Hicks M. Locksmith: Practical static race detection for c. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2011; **33**(1):3.

DOI: 10.1002/4345

20. Huang J, Zhang C. Persuasive prediction of concurrency access anomalies. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ACM, 2011; 144–154.
21. Chen J, MacDonald S. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*, ACM, 2008; 8.
22. Engler D, Ashcraft K. Racerx: effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review*, vol. 37, ACM, 2003; 237–252.
23. Lu K, Wu Z, Wang X, Chen C, Zhou X. Racechecker: efficient identification of harmful data races. *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, IEEE, 2015; 78–85.
24. Wu Z, Lu K, Wang X, Zhou X. Collaborative technique for concurrency bug detection. *International Journal of Parallel Programming* 2015; **43**(2):260–285.
25. Sen K. Race directed random testing of concurrent programs. *ACM SIGPLAN Notices* 2008; **43**(6):11–21.
26. Kasikci B, Zamfir C, Candea G. Racemob: crowdsourced data race detection. *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, ACM, 2013; 406–422.
27. Kasikci B, Zamfir C, Candea G. Data races vs data race bugs: telling the difference with portend. *ACM SIGPLAN Notices* 2012; **47**(4):185–198.
28. Zhang W, Sun C, Lu S. Conmem: detecting severe concurrency bugs through an effect-oriented approach. *ACM SIGARCH Computer Architecture News*, vol. 38, ACM, 2010; 179–192.
29. Microsoft. How to share memory between user mode and kernel mode 2015. URL https://support.microsoft.com/en-us/kb/191840.
30. Lewis N. Khobe attack technique: Kernel bypass risk or much ado about nothing? 2010. URL http://searchsecurity.techtarget.com/tip/KHOBE-attack-technique-Kernel-bypass-risk-or-much-ado-about-nothing.
31. Serna FJ. Ms08-061 : The case of the kernel mode double-fetch 2008. URL https://blogs.technet.microsoft.com/srd/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/.
32. Cox MJ. Bug 166248 - can-2005-2490 sendmsg compat stack overflow 2005. URL https://bugzilla.redhat.com/show_bug.cgi?id=166248.
33. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. *26th USENIX Security Symposium (USENIX Security 17)*, USENIX Association: Vancouver, BC, 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei.
34. Wang P. Double-fetch bug in linux-4.6/drivers/platform/chrome/cros_ec_dev.c 2016. URL https://bugzilla.kernel.org/show_bug.cgi?id=120131.
35. Wang P. Double-fetch bug in linux-4.5/drivers/s390/char/sclp_ctl.c 2016. URL https://bugzilla.kernel.org/show_bug.cgi?id=116741.
36. Prpic M. Cve-2016-6516 kernel: vfs: ioctl: double fetch leading to heap overflow 2016. URL https://bugzilla.redhat.com/show_bug.cgi?id=1362457.
37. Makelinux. I/o ports and i/o memory. http://www.makelinux.net/ldd3/chp-9-sect-1.
38. Makelinux. Using i/o memory. http://www.makelinux.net/ldd3/chp-9-sect-4.
39. Wiki. Memory-mapped i/o. https://en.wikipedia.org/wiki/Memory-mapped_I/O.
40. Hong K. Memory-mapped i/o vs port-mapped i/o - 2017. http://www.bogotobogo.com/Embedded/memory_mapped_io_vs_port_mapped_isolated_io.php.
41. Eckelmann S. [patch-resend] backports: Fix double fetch in hlist_for_each_entry*_rcu 2014. URL https://www.spinics.net/lists/backports/msg03072.html.
42. Wilhelm F. Xen xsa 155: Double fetches in paravirtualized devices 2015. URL https://www.insinuator.net/2015/12/xen-xsa-155-double-fetches-in-paravirtualized-devices/.
43. Wikipedia. Crash (computing) 2017. URL https://en.wikipedia.org/wiki/Crash_%28computing%29.
44. Exploiting windows drivers: Double-fetch race condition vulnerability 2016. URL http://resources.infosecinstitute.com/exploiting-windows-drivers-double-fetch-race-condition-vulnerability/.
45. Vulnerability caused by inconsistency checking 2011. URL https://maikiforever.wordpress.com/2011/10/07/.
46. Bauer S. Linux ¿= 4.5 double fetch leading to heap overflow 2016. URL http://www.openwall.com/lists/oss-security/2016/07/31/6.
47. Jurczyk M, Coldwind G. Bochspwn: Identifying 0-days via system-wide memory access pattern analysis. Black Hat 2013 2013. http://vexillium.org/dl.php?BH2013_Mateusz_Jurczyk_Gynvael_Coldwind.pdf.
48. Lawall J, Laurie B, Hansen RR, Palix N, Muller G. Finding error handling bugs in OpenSSL using Coccinelle. *European Dependable Computing Conference (EDCC)*, 2010; 191–196.
49. Volos H, Tack AJ, Swift MM, Lu S. Applying transactional memory to concurrency bugs. *ACM SIGPLAN Notices*, vol. 47, ACM, 2012; 211–222.
50. Pankratius V, Adl-Tabatabai AR, Otto F. *Does transactional memory keep its promises?: results from an empirical study*. Univ., Fak. für Informatik, 2009.
51. Rossbach CJ, Hofmann OS, Witchel E. Is transactional programming actually easier? *ACM Sigplan Notices* 2010; **45**(5):47–56.
52. Min SL, Choi JD. An efficient cache-based access anomaly detection scheme. *ACM SIGARCH Computer Architecture News* 1991; **19**(2):235–244.
53. Prvulovic M. Cord: Cost-effective (and nearly overhead-free) order-recording and data race detection. *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, IEEE, 2006; 232–243.

54. Prvulovic M, Torrellas J. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, IEEE, 2003; 110–121.
55. Zhou P, Teodorescu R, Zhou Y. Hard: Hardware-assisted lockset-based race detection. *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, IEEE, 2007; 121–132.
56. Muzahid A, Suárez D, Qi S, Torrellas J. Sigrace: signature-based data race detection. *ACM SIGARCH Computer Architecture News*, vol. 37, ACM, 2009; 337–348.
57. Erickson J, Musuvathi M, Burckhardt S, Olynyk K. Effective data-race detection for the kernel. *OSDI*, vol. 10, 2010; 1–16.

DOI: 10.1002/4345