usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel

Pengfei Wang, *National University of Defense Technology;* Jens Krinke, *University College London;* Kai Lu and Gen Li, *National University of Defense Technology;* Steve Dodier-Lazaro, *University College London*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei

This paper is included in the Proceedings of the
26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the
26th USENIX Security Symposium
is sponsored by USENIX

How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel

Pengfei Wang
National University of Defense Technology

Jens Krinke
University College London

Kai Lu
National University of Defense Technology

Gen Li
National University of Defense Technology

Steve Dodier-Lazaro
University College London

Abstract

We present the first static approach that systematically detects potential double-fetch vulnerabilities in the Linux kernel. Using a pattern-based analysis, we identified 90 double fetches in the Linux kernel. 57 of these occur in drivers, which previous dynamic approaches were unable to detect without access to the corresponding hardware. We manually investigated the 90 occurrences, and inferred three typical scenarios in which double fetches occur. We discuss each of them in detail. We further developed a static analysis, based on the Coccinelle matching engine, that detects double-fetch situations which can cause kernel vulnerabilities. When applied to the Linux, FreeBSD, and Android kernels, our approach found six previously unknown double-fetch bugs, four of them in drivers, three of which are exploitable double-fetch vulnerabilities. All of the identified bugs and vulnerabilities have been confirmed and patched by maintainers. Our approach has been adopted by the Coccinelle team and is currently being integrated into the Linux kernel patch vetting. Based on our study, we also provide practical solutions for anticipating double-fetch bugs and vulnerabilities. We also provide a solution to automatically patch detected double-fetch bugs.

1 Introduction

The wide use of multi-core hardware is making concurrent programs increasingly pervasive, especially in operating systems, real-time systems and computing intensive systems. However, concurrent programs are also notorious for difficult to detect concurrency bugs. Real-world concurrency bugs can be categorized into three types: atomicity-violation bugs, order-violation bugs, and deadlocks [20].

A data race is another common situation in concurrent programs. It occurs when two threads are accessing one shared 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 [30, 15]. Data races usually lead to concurrency bugs because they can cause atomicity-violations [22, 21, 23] or order-violations [33, 40]. In addition to occurring between two threads, data races can also happen across the kernel and user space. Serna [32] was the first to use the term “double fetch” to describe a Windows kernel vulnerability due to a race condition in which the kernel fetches the same user space data twice. A double-fetch bug occurs when the kernel reads and uses the same value that resides in the user space twice (expecting it to be identical both times), while a concurrently running user thread can modify the value in the time window between the two kernel reads. Double-fetch bugs introduce data inconsistencies in the kernel code, leading to exploitable vulnerabilities such as buffer overflows [1, 32, 14, 37].

Jurczyk and Coldwind [14] were the first to study double fetches systematically. Their dynamic approach detected double fetches by tracing memory accesses and they discovered a series of double-fetch vulnerabilities in the Windows kernel. However, their dynamic approach can achieve only limited coverage. In particular, it cannot be applied to code that needs corresponding hardware to be executed, so device drivers cannot be analyzed without access to the device or a simulation of it. Thus, their analysis cannot cover the entirety of the kernel. In fact, their approach has not discovered any double-fetch vulnerability in Linux, FreeBSD or OpenBSD [13]. Besides, Jurczyk and Coldwind have brought attention to not only on how to find but also on *how to exploit* double-fetch vulnerabilities. Instructions on how to exploit double fetches have recently become publicly available [11]. Thus, auditing kernels, in particular drivers, for double-fetch vulnerabilities has become urgent.

Device drivers are critical kernel-level programs that bridge hardware and software by providing interfaces between the operating system and the devices attached to the system. Drivers are a large part of current operating systems, e.g., 44% of the Linux 4.5 source files belong to drivers. Drivers were found to be particularly bug-prone kernel components. Chou et al. [7] empirically showed that the error-rate in device drivers is about ten times higher than in any other parts of the kernel. Swift et al. [34] also found that 85% of system crashes in Windows XP can be blamed on driver errors. Furthermore, Ryzhyk et al. [29] found that 19% of the bugs in drivers were concurrency bugs, and most of them were data races or deadlocks.

Because drivers are such a critical point of failure in kernels, they must be analyzed for security vulnerabilities even when their corresponding hardware is not available. Indeed, 26% of the Linux kernel source files belong to hardware architectures other than x86 which cannot be analyzed with Jurczyk and Coldwind's x86-based technique. Thus, dynamic analysis is not a viable, affordable approach. Therefore, we developed a static pattern-based approach to identify double fetches in the Linux kernel, including the complete space of drivers. We identified 90 double fetches which we then investigated and categorized into three typical scenarios in which double fetches occur. We found that most double fetches are not double-fetch bugs because although the kernel *fetches* the same data twice, it only *uses* the data from one of the two fetches. We therefore refined the static pattern-based approach to detect actual double-fetch bugs and vulnerabilities, and analyzed Linux, Android and FreeBSD with it.

We found that most of the double fetches in Linux 4.5 occur in drivers (57/90) and so do most of the identified double-fetch bugs (4/5). This means dynamic analysis methods fail to detect a majority of double fetch bugs, unless researchers have access to the complete range of hardware compatible with the kernel they analyze. This is confirmed by a comparison with Bochspwn, a dynamic analysis approach, which was unable to find any double-fetch bug in Linux 3.5.0 [13] where our approach finds three. In summary, we make the following contributions in this paper:

(1) First systematic study of double fetches in the Linux kernel. We present the first (to the best of our knowledge) study of double fetches in the complete Linux kernel, including an analysis of how and why a double fetch occurs. We used pattern matching to automatically identify 90 double-fetch situations in the Linux kernel, and investigated those candidates by manually reviewing the kernel source. We categorize the identified double fetches into three typical scenarios (*type selection*, *size checking*, *shallow copy*) in which double fetches are prone to occur, and illustrate each scenario with a detailed double fetch case analysis. Most (57/90) of the identified double fetches occur in drivers.

(2) A pattern-based double-fetch bug detection approach. We developed a static pattern-based approach to detect double-fetch bugs[1]. The approach has been implemented on the Coccinelle program matching and transformation engine [17] and has been adapted for checking the Linux, FreeBSD, and Android kernels. It is the first approach able to detect double-fetch vulnerabilities in the complete kernel including all drivers and all hardware architectures. Our approach has been adopted by the Coccinelle team and is currently being integrated into the Linux kernel patch vetting through Coccinelle.

(3) Identification of six double-fetch bugs. In total, we found six real double-fetch bugs. Four are in the drivers of Linux 4.5 and three of them are exploitable vulnerabilities. Moreover, all four driver-related double-fetch bugs belong to the same *size checking* scenario. The bugs have been confirmed by the Linux maintainers and have been fixed in new versions as a result of our reports. One double-fetch vulnerability has been found in the Android 6.0.1 kernel, which was already fixed in newer Linux kernels.

(4) Strategies for double-fetch bug prevention. Based on our study, we propose five solutions to anticipate double-fetch bugs and we implemented one of the strategies in a tool that automatically patches double-fetch bugs.

The rest of the paper is organized as follows: Section 2 presents relevant background on memory access in Linux, specifically in Linux drivers, and on how double-fetch bugs occur. Section 3 introduces our approach to double fetch detection, including our analysis process, the categorization of the identified double fetches into three scenarios, and what we learned from the identified double-fetch bugs. Section 4 presents the evaluation of our work, including statistics on the manual analysis and the results of applying our approach to the Linux, FreeBSD, and Android kernels. Section 5 discusses the detected bugs, implications of double-fetch bug prevention, an interpretation of our findings, as well as limitations of our approach. Related work is discussed in Section 6, followed by conclusions.

2 Background

We provide readers with a reminder of how data is exchanged between the Linux kernel and its drivers and the user space, and of how race conditions and double-fetch bugs can occur within this framework.

[1] Our analysis is available at `https://github.com/UCL-CREST/doublefetch`

2.1 Kernel/User Space Protection

In modern computer systems, memory is divided into kernel space and user space. The kernel space is where the kernel code executes and where its internal data is stored, while the user space is where normal user processes run. Each user space process resides in its own address space, and can only address memory within that space. Those virtual address spaces are mapped onto physical memory by the kernel in such a way that isolation between separate spaces is guaranteed. The kernel also has its own independent address space.

Special schemes are provided by the operating system to exchange data between kernel and user space. In Windows, we can use the device input and output control (IOCTL) method, or a shared memory object method to exchange data between kernel and user space[2] which is very similar to shared memory regions. In Linux and FreeBSD, functions are provided to safely transfer data between kernel space and user space which we call *transfer functions*. For instance, Linux has four often used transfer functions, `copy_from_user()`, `copy_to_user()`, `get_user()`, and `put_user()`, that copy single values or an arbitrary amount of data to and from user space in a safe way. Transfer functions not only exchange data between kernel and user space but also provide a protection mechanism against invalid memory access, such as illegal addresses or page faults. Therefore, any double fetch in Linux will involve multiple invocations of transfer functions.

2.2 Memory Access in Drivers

Device drivers are kernel components responsible for enabling the kernel to communicate with and make use of hardware devices connected to the system. Drivers have typical characteristics, such as support for synchronous and asynchronous operations and the ability to be opened multiple times [8]. Drivers are critical to security because faults in them can result in vulnerabilities that grant control of the whole system. Finally, drivers often have to copy messages of variable type or variable length from the user space to the hardware, and, as we will see later, this often leads to double-fetch situations that cause vulnerabilities.

In Linux, all devices have a file representation which can be accessed from user space to interact with the hardware's driver. The kernel creates a file in the `/dev` directory for each driver, with which user space processes can interact using file input/output system calls. The driver provides implementations of all file related operations, including `read()` and `write()` functions. In such functions, the driver needs to fetch the data from the user space (in `write`) or copy data to the user space (in `read`). The driver uses the transfer functions to do so, and again, any double fetch will involve multiple invocations of transfer functions.

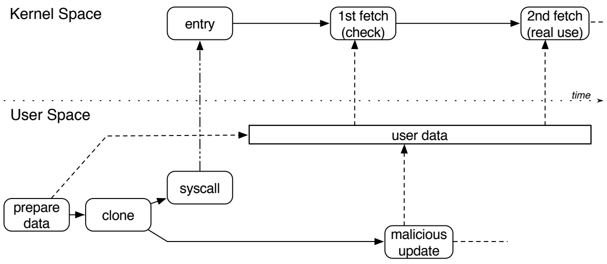


Figure 1: Principal Double Fetch Race Condition

2.3 Double Fetch

A double fetch is a special case of a race condition that occurs in memory access between the kernel and user space. The first vulnerability of this type was presented by Serna [32] in a report on Windows double-fetch vulnerabilities. Technically, a double fetch takes place within a kernel function, such as a syscall, which is invoked by a user application from user mode. As illustrated in Figure 1, the kernel function fetches a value twice from the same memory location in the user space, the first time to check and verify it and the second time to use it (note that the events are on a timeline from left to right, but the user data is the same object all the time). Meanwhile, within the time window between the two kernel fetches, a concurrently running user thread modifies the value. Then, when the kernel function fetches the value a second time to use, it gets a different value, which will not only result in a different computation outcome, but may cause a buffer overflow, a null-pointer crash or even worse consequences.

To avoid confusion, we use the term *double fetch* or *double-fetch situation* in this paper to represent all the situations in which the kernel fetches the same user data more than once, and a so-called double fetch can be further divided into the following cases:

Benign double fetch: A benign double fetch is a case that will not cause harm, owing to additional protection schemes or because the double-fetched value is not used twice (details will be discussed in Section 5.3).

Harmful double fetch: A harmful double fetch or a *double-fetch bug* is a double fetch that could actually cause failures in the kernel in specific situations, e.g., a race condition that could be triggered by a user process.

Double-fetch vulnerability: A double-fetch bug can also turn into a *double-fetch vulnerability* once the consequence caused by the race condition is exploitable, such

[2] https://support.microsoft.com/en-us/kb/191840

```
140 int cmsghdr_from_user_compat_to_kern(struct msghdr *kmsg,
141               unsigned char *stackbuf, int stackbuf_size)
142 {
143   struct compat_cmsghdr __user *ucmsg;
144   struct cmsghdr *kcmsg, *kcmsg_base;
145   compat_size_t ucmlen;
...
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((...)(((char __user *)ucmsg - (char __user*)...
160                        + ucmlen) > kmsg->msg_controllen)
161       return -EINVAL;
...
166     ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
167   }
168   if(kcmlen == 0)
169     return -EINVAL;
...
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),
194             CMSG_COMPAT_DATA(ucmsg),
195             (ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg)))))
...
212 }
```

Figure 2: Double-Fetch Vulnerability in Linux 2.6.9

as through a buffer overflow, causing privilege escalation, information leakage or kernel crash.

In this paper, we investigate both harmful double fetches and benign double fetches. Even though benign double fetches are currently not vulnerable, some of them can turn into harmful ones when the code is changed or updated in the future (when the double-fetched data is reused). Moreover, some benign double fetches them can cause performance degradation when one of the fetches is redundant (discussed in Section 5).

Double-fetch vulnerabilities occur not only in the Windows kernel [14], but also in the Linux kernel. Figure 2 shows a double-fetch bug in Linux 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 a sanity check at the second time. This can cause a kernel buffer overflow and therefore could lead to a privilege escalation. The 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 checked (lines 156–161) before use, whereas after the second fetch (line 185) there are no checks before use, which may cause an overflow in the copy operation (line 195) that can be exploited to execute arbitrary code by modifying the message.

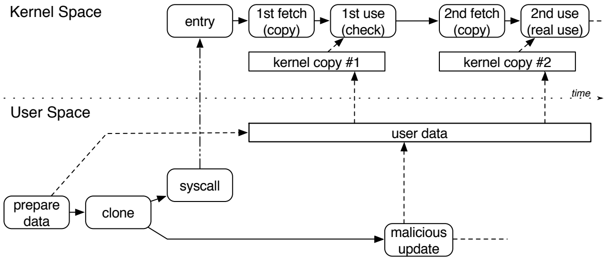


Figure 3: Double Fetch with Transfer Functions

Plenty of approaches have been proposed for data race detection at memory access level. Static approaches analyze the program without running it [35, 28, 12, 6, 10, 19, 38]. However, their major disadvantage is that they generate a large number of false reports due to lack the full execution context of the program. Dynamic approaches execute the program to verify data races [31, 16, 15], checking whether a race could cause a program failure in executions. Dynamic approaches usually control the active thread scheduler to trigger specific interleavings to increase the probability of a bug manifestation [41]. Nevertheless, the runtime overhead is a severe problem and testing of driver code requires the support of specific hardware or a dedicated simulation. Unfortunately, none of the existing data race detection approaches (whether static or dynamic) can be applied to double-fetch bug detection directly, for the following reasons:

(1) A double-fetch bug is caused by a race condition between kernel and user space, which is different from a common 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, and most of the previous approaches detect data races by identifying *all* read and write operations accessing the same memory location. However, things are different for a double-fetch bug. The kernel only contains two reads while the write resides in the user thread. Moreover, the double-fetch bug exists if there is a possibility that the kernel fetches and uses the same memory location twice, as a malicious user process can specifically be designed to write between the first and second fetch.

(2) The involvement of the kernel makes a double-fetch bug different from a data race in the way of accessing data. In Linux, fetching data from user space to kernel space relies on the specific parameters passed to transfer functions (e.g., `copy_from_user()` and `get_user()`) rather than dereferencing the user pointer directly, which means the regular data race detection approaches based on pointer dereference are not applicable anymore.

(3) Moreover, a double-fetch bug in Linux is more complicated than a common data race or a double-fetch bug in Windows. As shown in Figure 3, a double-fetch bug in Linux requires a first fetch that copies the data, usually followed by a first check or use of the copied data, then a second fetch that copies the same data again, and a second use of the same data. Although the double fetch can be located by matching the patterns of fetch operations, the use of the fetched data varies a lot. For example, in addition to being used for validation, the first fetched value can be possibly copied to somewhere else for later use, which means the first use (or check) could be temporally absent. Besides, the fetched value can be passed as an argument to other functions for further use. Therefore, in this paper, we define the *use* in a double fetch to be a conditional check (read data for comparison), an assignment to other variables, a function call argument pass, or a computation using the fetched data. We need to take into consideration these double fetch characteristics.

For these reasons, identifying double-fetch bugs requires a dedicated analysis and previous approaches are either not applicable or not effective.

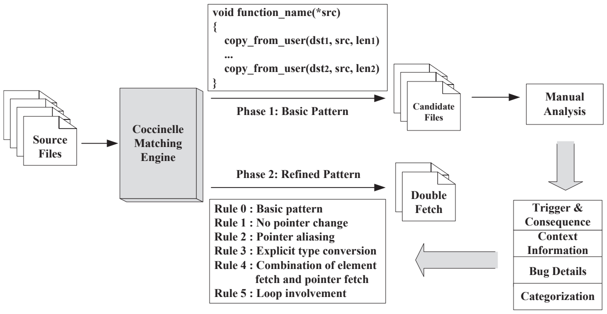
2.4 Coccinelle

Coccinelle [17] is a program matching and transformation engine with a dedicated language SmPL (Semantic Patch Language) for specifying desired matches and transformations in C code. Coccinelle was initially targeted for collateral evolution in Linux drivers, but now is widely used for finding and fixing bugs in systems code.

Coccinelle's strategy for traversing control-flow graphs is based on temporal logic CTL (Computational Tree Logic) [3], and the pattern matching implemented on Coccinelle is path-sensitive, which achieves better code coverage. Coccinelle is highly optimized to improve performance when exhaustively traversing all the execution paths. Besides, Coccinelle is insensitive to newlines, spaces, comments, etc. Moreover, the pattern-based analysis is applied directly to the source code, therefore operations that are defined as macros, such as `get_user()` or `__get_user()`, will not be expanded during the matching, which facilitates the detection of double fetches based on the identification of transfer functions. Therefore, Coccinelle is a suitable tool for us to carry out our study of double fetches based on pattern matching.

3 Double Fetches in the Linux Kernel

In this paper, our study of double fetches in the Linux kernel is divided into two phases. As shown in Figure 4, in the first phase, we analyze the Linux kernel with the Coccinelle engine using a basic double-fetch pattern that identifies when a function has multiple invocations of a transfer function. Then we manually investigate the candidate files found by the pattern matching, to categorize the scenarios in which a double fetch occurs and when a double-fetch bug or vulnerability is prone to happen based on the context information that is relevant to the bug. In the second phase, based on the knowledge gained from the manual analysis, we developed a more precise analysis using the Coccinelle engine to systematically detect double-fetch bugs and vulnerabilities throughout the kernel, which we also used to additionally analyze FreeBSD and Android.



```
void function_name(*src)
{
   copy_from_user(dst1, src, len1)
   ...
   copy_from_user(dst2, src, len2)
}
```

Source Files
Coccinelle Matching Engine
Phase 1: Basic Pattern
Candidate Files
Manual Analysis
Phase 2: Refined Pattern
Double Fetch
Rule 0 : Basic pattern
Rule 1 : No pointer change
Rule 2 : Pointer aliasing
Rule 3 : Explicit type conversion
Rule 4 : Combination of element fetch and pointer fetch
Rule 5 : Loop involvement
Trigger & Consequence
Context Information
Bug Details
Categorization

Figure 4: Overview of our Two-Phase Coccinelle-Based Double-Fetch Situation Detection Process

3.1 Basic Pattern Matching Analysis

There are situations in which a double fetch is hard to avoid, and there exist a large number of functions in the Linux kernel that fetch the same data twice. According to the definition, a double fetch can occur in the kernel when the same user data is fetched twice within a short interval. Therefore we can conclude a basic pattern that we will use to match all the potential double-fetch situations. The pattern matches the situation in which a kernel function is using transfer functions to fetch data from same user memory region at least twice. In the case of the Linux kernel, the transfer functions to match are mainly `get_user()` and `copy_from_user()` in all their variants. The pattern allows the target of the copy and the size of the copied data to be different, but the source of copy (the address in user space) must be the same. As shown in Figure 4, we implemented the basic pattern matching in the Coccinelle engine.

Our approach examines all source code files of the Linux kernel and checks whether a kernel function contains two or more invocations of transfer functions that fetch data from the same user pointer. From the 39,906 Linux source files, 17,532 files belong to drivers (44%), and 10,398 files belong to non-x86 hardware architectures (26%) which cannot be analyzed with Jurczyk and Coldwind's x86-based technique. We manually analyzed the matched kernel functions to infer knowledge on the characteristics of double fetches, i.e., how the user data is transferred to and used in the kernel, which helped us to carry out a categorization of double-fetch situations, as we discuss in Section 3.2. The manual analysis also helped us refine our pattern matching approach and more precisely detect actual double-fetch bugs, as explained in Section 3.3.

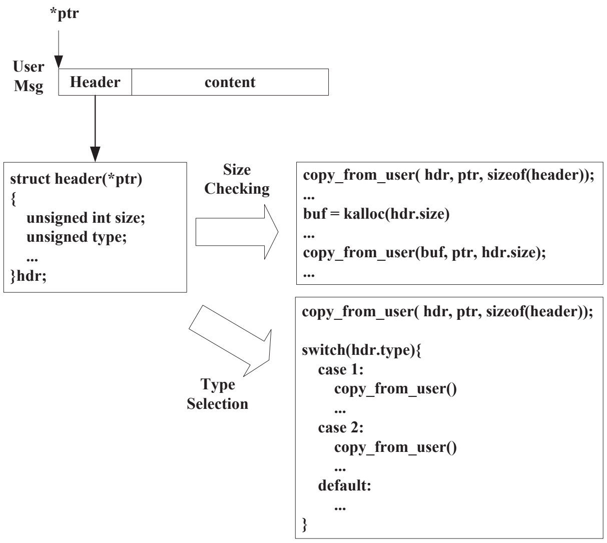
During the investigation, we noticed that there are plenty of cases where the transfer functions fetch data from different addresses or from the same address but with different offsets. For example, a kernel function may fetch the elements of a specific structure separately instead of copying the whole structure to the kernel. By adding different offsets to the start address of that structure, the kernel fetches different elements of the structure separately, which results in multiple fetches. Another common situation is adding a fixed offset to the source pointer, so as to process a long message separately, or just using self-increment (++) to process a message automatically in a loop. All these cases are false positives caused by the basic pattern matching, and 226 cases of our initial reports were identified as false positives, which have been automatically removed in our refined phase since they are not considered as double-fetch situations and cannot cause a double-fetch bug because every single piece of the message is only fetched once.

The first phase of our study concentrates on the understanding of the contexts in which double fetches are prone to happen, rather than on exhaustively finding potential double-fetch bugs. Even though the analysis and characterization is not fully automated, it only resulted in 90 candidates that needed manual investigation, which took only a few days to analyze them, making the needed manual effort of our approach acceptable.

3.2 Double Fetch Categorization

As we manually inspected the double fetch candidates, we noticed that there are three common scenarios in which double fetches are prone to happen, which we categorized as *type selection*, *size checking* and *shallow copy*. We now discuss these in detail.

Most of the time, copying data from the user space to the kernel space is straightforward via a single invocation of a transfer function. However, things get complicated when the data has a variable type or a variable length, depending on the data itself. Such data usually starts with a *header*, followed by the data's *body*. In the following, we consider such data to be *messages*, as we empirically found that variable data was often used by drivers to pass messages to the hardware from user space.



```
struct header(*ptr)
{
   unsigned int size;
   unsigned type;
   ...
}hdr;
```

```
copy_from_user( hdr, ptr, sizeof(header));
...
buf = kalloc(hdr.size)
...
copy_from_user(buf, ptr, hdr.size);
...
```

```
copy_from_user( hdr, ptr, sizeof(header));

switch(hdr.type){
   case 1:
      copy_from_user()
      ...
   case 2:
      copy_from_user()
      ...
   default:
      ...
}
```

Figure 5: How Message Structure Leads to Double Fetches

Figure 5 illustrates the scenario: A message copied from the user space to the kernel (driver) space usually consists of two parts, the header and the body. The header contains some meta information about the message, such as an indicator of the message *type* or the *size* of the message body. Since messages have different types and the message lengths may also vary, the kernel usually fetches (copies) the header first to decide which buffer type needs to be created or how much space needs to be allocated for the complete message. A second fetch then copies the complete message into the allocated buffer of the specified type or size. The second fetch not only copies the body, but also copies the complete message including the header which has been fetched already. Because the header of the message is fetched (copied) twice, a double-fetch situation arises. The double-fetch situation turns into a double-fetch bug when the size or type information from the **second** fetch is used as the user may have changed the size or type information between the two fetches. If, for example, the size information is used to control buffer access, the double-fetch bug turns into a vulnerability.

The double-fetch situations where a message header is copied twice could easily be avoided by only copying the message body in the second fetch and then joining the header with the body. However, copying the complete message in the second step is more convenient, and therefore such a double-fetch situation occurs very often in the Linux kernel. Moreover, large parts of the Linux kernel are old, i.e., they have been developed before double-fetch bugs were known or understood. Therefore, we will discuss such double-fetch situations in the kernel in more detail and also highlight three cases we have found during the manual analysis.

3.2.1 Type Selection

A common scenario in which double fetches occur is when the message header is used for type selection. In other words, the header of the message is fetched first to recognize the message type and then the whole message is fetched and processed dependent on the identified type. We have observed that it is very common in the Linux kernel that one single function in a driver is designed to handle multiple types of messages by using a `switch` statement structure, in which each particular message type is fetched and then processed. The result of the first fetch (the message type) is used in the `switch` statement's condition and in every case of the `switch`, the message is then copied by a second fetch to a local buffer of a specific type (and then processed).

Figure 6 shows an example of a double-fetch situation due to type selection in the file `cxgb3_main.c`, part of a network driver. The function `cxgb_extension_ioctl()` first fetches the type of the message (a command for the attached hardware) into `cmd` from the pointer into user space `useraddr` at line 2136. It then decides based on `cmd` which type the message has (e.g., `CHELSIP_SET_QSET_PARAMS`, `CHELSIP_-SET_QSET_NUM` or `CHELSIO_SETMTUTAB`) and copies the complete message into the corresponding structure (of type `ch_qset_params`, `ch_reg`, `ch_mtus`, ...). The type of the message will be fetched a second time as part of the whole message (lines 2149, 2292, 2355 respectively). As long as the header part of the message is not used again, the double fetch in this situation does not cause a double-fetch bug. However, if the header part (the type/command) of the second fetch would be used again, problems could occur as a malicious user could have changed the header between the two fetches. In the case of `cxgb_extension_ioctl()`, a manual investigation revealed no use of the type part in the buffers `t`, `edata`, `m`, ... and the double-fetch situation here does not cause a double-fetch vulnerability.

We found 11 occurrences of this double-fetch category, 9 of them in drivers. None of the 11 occurrences used the header part of the second fetch and therefore, they were not causing double-fetch bugs.

```
2129 static int cxgb_extension_ioctl(struct net_device *dev,
                                 void __user *useraddr)
2130 {
...
2133   u32 cmd;
...
2136   if (copy_from_user(&cmd, useraddr, sizeof(cmd)))
2137     return -EFAULT;
2138
2139   switch (cmd) {
2140   case CHELSIO_SET_QSET_PARAMS:{
...
2143     struct ch_qset_params t;
...
2149     if (copy_from_user(&t, useraddr, sizeof(t)))
2150       return -EFAULT;
2151     if (t.qset_idx >= SGE_QSETS)
2152       return -EINVAL;
...
2238     break;
2239   }
...
2284   case CHELSIO_SET_QSET_NUM:{
2285     struct ch_reg edata;
...
2292     if (copy_from_user(&edata, useraddr, sizeof(edata)))
2293       return -EFAULT;
2294     if (edata.val < 1 ||
2295         (edata.val > 1 && !(...)))
2296       return -EINVAL;
...
2313     break;
2314   }
...
2345   case CHELSIO_SETMTUTAB:{
2346     struct ch_mtus m;
...
2355     if (copy_from_user(&m, useraddr, sizeof(m)))
2356       return -EFAULT;
2357     if (m.nmtus != NMTUS)
2358       return -EINVAL;
2359     if (m.mtus[0] < 81)
2360       return -EINVAL;
...
2369     break;
2370   }
...
2499 }
```

Figure 6: A Double-Fetch Situation Belonging to the Type Selection Category in `cxgb3_main.c`

3.2.2 Size Checking

Another common scenario occurs when the actual length of the message can vary. In this scenario, the message header is used to identify the size of the complete message. The message header is copied to the kernel first to get the message size (first fetch), check it 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. As long as only the size of the first fetch is used and not retrieved from the header of the second fetch, the double fetch in this situation does not cause a double-fetch vulnerability or bug. However, if the size is retrieved from the header of the second fetch and used, the kernel becomes vulnerable as a malicious user could have changed the size element of the header.

One such double-fetch bug (CVE-2016-6480) was found in file `commctrl.c` in the Adaptec RAID controller driver of the Linux 4.5. Figure 7 shows the responsible function `ioctl_send_fib()` which fetches data from user space pointed by pointer `arg` via `copy_from_user()` twice in line 81 and line 116. The

```
 60 static int ioctl_send_fib(struct aac_dev* dev,
                             void __user *arg)
 61 {
 62   struct hw_fib * kfib;
...
 81   if (copy_from_user((void *)kfib, arg, sizeof(...))) {
 82     aac_fib_free(fibptr);
 83     return -EFAULT;
 84   }
...
 90   size = le16_to_cpu(kfib->header.Size) + sizeof(...);
 91   if (size < le16_to_cpu(kfib->header.SenderSize))
 92     size = le16_to_cpu(kfib->header.SenderSize);
 93   if (size > dev->max_fib_size) {
...
101     kfib = pci_alloc_consistent(dev->pdev, size, &daddr);
...
114   }
115
116   if (copy_from_user(kfib, arg, size)) {
117     retval = -EFAULT;
118     goto cleanup;
119   }
120
121   if (kfib->header.Command == cpu_to_le16(...)) {
...
128   } else {
129     retval =
        aac_fib_send(le16_to_cpu(kfib->header.Command),...
130            le16_to_cpu(kfib->header.Size) , FsaNormal,
131            1, 1, NULL, NULL);
...
139   }
...
160 }
```

Figure 7: A Double-Fetch Vulnerability in `commctrl.c` (CVE-2016-6480)

first fetched value is used to calculate a buffer size (line 90), to check the validity of the size (line 93), and to allocate a buffer of the calculated size (line 101), while the second copy (line 116) fetches the whole message with the calculated size. Note that the variable `kfib` pointing to the kernel buffer storing the message is reused in line 101. The header of the message is large and various elements of the header are used after the message has been fetched the second time (e.g., `kfib->header.Command` in line 121 and 129). The function also uses the size element of the header a second time in line 130, causing a double-fetch vulnerability as a malicious user could have changed the `Size` field of the header between the two fetches.

We observed 30 occurrences of such size checking double-fetch situations, 22 of which occur in drivers, and four of them (all in drivers) are vulnerable.

3.2.3 Shallow Copy

The last special case of double-fetch scenario we identified is what we call *shallow copy issues*. A shallow copy between user and kernel space happens when a buffer (the first buffer) in the user space is copied to the kernel space, and the buffer contains a pointer to another buffer in user space (the second buffer). A transfer function only copies the first buffer (a shallow copy) and the second buffer has to be copied by the second invocation of a transfer function (to perform a deep copy). Sometimes it is necessary to copy data from user space into kernel space, act on the data, and copy the data back into user space. Such data is usually contained in the second buffer in user space and pointed to by a pointer in the first buffer in user space containing additional data. The transfer functions perform shallow copies and therefore data pointed to in the buffer copied by a transfer function must be explicitly copied as well, so as to perform a deep copy. Such deep copies will cause multiple invocations of transfer functions which are not necessarily double fetches as each transfer function is invoked with a different buffer to be copied. We observed 31 of such situations, 19 of them in drivers.

```
55 static int sclp_ctl_ioctl_sccb(void __user *user_area)
56 {
57   struct sclp_ctl_sccb ctl_sccb;
58   struct sccb_header *sccb;
59   int rc;
60
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   }
72   if (sccb->length > PAGE_SIZE || sccb->length < 8)
73     return -EINVAL;
74   if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                        sccb->length)) {
75     rc = -EFAULT;
76     goto out_free;
77   }
...
81   if (copy_to_user(u64_to_uptr(ctl_sccb.sccb), sccb,
                     sccb->length))
82     rc = -EFAULT;
...
86 }
```

Figure 8: A Double-Fetch Bug in `sclp_ctl.c` (CVE-2016-6130)

The complexity of performing a deep copy with transfer functions that only do shallow copies can cause programmers to introduce bugs, and we found one such bug in file `sclp_ctl.c` of the IBM S/390 SCLP console driver, where the bug is caused by a shallow copy issue (CVE-2016-6130). The function `sclp_ctl_ioctl_sccb` in Figure 8 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 then has to copy another data structure from user space pointed to by `ctl_sccb.sccb`. However, the size of the

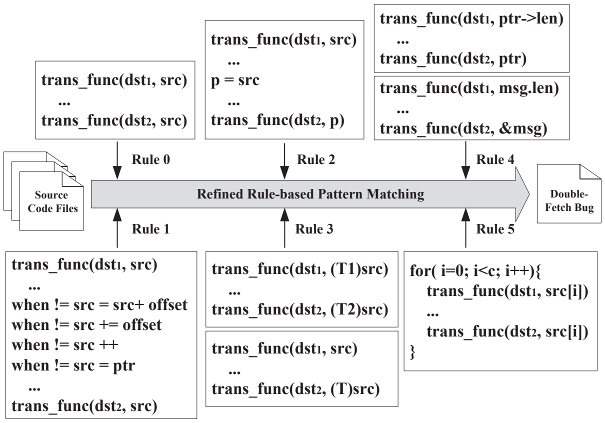


Figure 9: Refined Coccinelle-Based Double-Fetch Bug Detection

data structure is variable, causing a *size checking* scenario. In order to copy the 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` which is checked for validity in line 72. Then, based on `sccb->length`, it copies the whole content with a second fetch in line 74. Finally at line 81, the data is copied back to the user space. While it looks like both invocations of the transfer functions in lines 74 and 81 use the same length `sccb->length`, line 81 actually uses the value as copied in line 74 (the second fetch) while line 74 uses the value from the first fetch.

Again, this is a double-fetch bug as a user may have changed the value between the two fetches in lines 68 and 74. However, this double-fetch bug is not causing a vulnerability because neither can the kernel be crashed by an invalid size given to a transfer function, nor can information leakage occur when the kernel copies back data beyond the size that it received earlier because the copied buffer is located in its own memory page. An attempt to trigger the bug will simply end in termination of the system call with an error code in line 82. The double-fetch bug has been eliminated in Linux 4.6.

3.3 Refined Double-Fetch Bug Detection

In this section, we present the second phase of our study which uses a refined double-fetch bug detection approach that is again based on the Coccinelle matching engine. While the first phase of our study was to identify and categorize scenarios in which double fetches occur, the second phase exploited the gained knowledge from the first phase to design an improved analysis targeted at specifically identifying double-fetch bugs and vulnerabilities.

As shown in Figure 9, in addition to the basic double-fetch pattern matching rule (**Rule 0**), which is triggered when two reads fetch data from the same source location, we added the following five additional rules to improve precision as well as discover corner cases. The Coccinelle engine applies these rules one by one when analyzing the source files. A double-fetch bug could involve different transfer functions, therefore, we have to take the four transfer functions that copy data from user space (`get_user()`, `__get_user()`, `copy_from_user()`, `__copy_from_user()`) into consideration. We use `trans_func()` in Figure 9 to represent any possible transfer functions in the Linux kernel.

Rule 1: No pointer change. The most critical rule in detecting double-fetch bugs is keeping the user pointer unchanged between two fetches. Otherwise, different data is fetched each time instead of the same data being double-fetched, and false positives can be caused. As can be seen from Rule 1 in Figure 9, this change might include cases of self-increment (++), adding an offset, or assignment of another value, and the corresponding subtraction situations.

Rule 2: Pointer aliasing. Pointer aliasing is common in double-fetch situations. In some cases, the user pointer is assigned to another pointer, because the original pointer might be changed (e.g., processing long messages section by section within a loop), while using two pointers is more convenient, one for checking the data, and the other for using the data. As can be seen from Rule 2 in Figure 9, this kind of assignment might appear at the beginning of a function or in the middle between the two fetches. Missing aliasing situation could cause false negatives.

Rule 3: Explicit type conversion. Explicit pointer type conversion is widely used when the kernel is fetching data from user space. For instance, in the size checking scenario, a message pointer would be converted to a header pointer to get the header in the first fetch, then used again as a message pointer in the second fetch. As can be seen from Rule 3 in Figure 9, any of the two source pointers could involve type conversion. Missing type conversion situations could cause false negatives. In addition, explicit pointer type conversions are usually combined with pointer aliasing, causing the same memory region to be manipulated by two types of pointers.

Rue 4: Combination of element fetch and pointer fetch. In some cases, a user pointer is used to both fetch the whole data structure as well as fetching only a part by dereferencing the pointer to an element of the data structure. For instance, in the size checking scenario, a user pointer is first used to fetch the message length by `get_user(len, ptr->len)`, then to copy the whole message in the second fetch by `copy_from_user(msg, ptr, len)`, which means the two fetches are not using exactly the same pointer as the transfer function arguments, but they cover the same value semantically. As we can see from Rule 4 in Figure 9, this situation may use a user pointer or the address of the data structure as the argument of the transfer functions. This situation usually appears with explicit pointer type conversion, and false negatives could be caused if this situation is missed.

Rule 5: Loop involvement. Since Coccinelle is path-sensitive, when a loop appears in the code, one transfer function call in a loop will be reported as two calls, which could cause false positives. Besides, as can be seen from Rule 5 in Figure 9, when there are two fetches in a loop, the second fetch of the last iteration and the first fetch of the next iteration will be matched as a double fetch. This case should be removed as false positive because the user pointer should have been changed when crossing the iterations and these two fetches are getting different values. Moreover, cases that use an array to copy different values inside a loop also cause false positives.

4 Evaluation

In this section, we present the evaluation of our study, which includes two parts: the statistics of the manual analysis, and the results of the refined approach when applied to three open source kernels: Linux, Android, and FreeBSD. We obtained the most up-to-date versions available at the time of the analysis.

4.1 Statistics and Analysis

In Linux 4.5, there are 52,881 files in total and 39,906 of them are source files (with a file extension of .c or .h), which are our analysis targets (other files are ignored). 17,532 source files belong to drivers (44%). After the basic pattern matching of the source files and the manual inspection to remove false positives, we obtained 90 double-fetch candidate files for further inspection. We categorized the candidates into the three double-fetch scenarios *Size Checking*, *Type Selection* and *Shallow Copy*. They are the most common cases on how a double fetch occurs while user space data is copied to the kernel space and how the data is then used in the kernel. We have discussed these scenarios in detail with real double-fetch bug examples in the previous section. As shown in Table 1, of the 90 candidates we found, 30 were related to the size checking scenario, 11 were related to the type selection scenario, and 31 were related to the shallow copy scenario, accounting for 33%, 12%, and 34% respectively. 18 candidates did not fit into one of the three scenarios.

Furthermore, 57 out of the 90 candidates were part of Linux drivers and among them, 22 were size checking related, 9 were type selection related and 19 were shallow copy related.

Table 1: Basic Double Fetch Analysis Results

Category	Occurrences		In Drivers	
Size Checking	30	33%	22	73%
Type Selection	11	12%	9	82%
Shallow Copy	31	34%	19	61%
Other	18	20%	7	39%
Total	90	100%	57	63%
True Bugs	5	6%	4	80%

Table 2: Refined Double-Fetch Bug Detection Results

Kernel	Total Files	Reported Files	True Bugs	Size Check.	Type Sel.
Linux 4.5	39,906	53	5	23	6
Android 6.0.1	35,313	48	3	18	6
FreeBSD	32,830	16	0	8	3

Most importantly, we found five previously unknown double-fetch bugs which include four size checking scenarios and one shallow copy scenario which also belongs to the size checking scenario. Three of them are exploitable vulnerabilities. The five bugs have been reported and they all have been confirmed by the developers and have meanwhile been fixed. From the statistical result, we can observe the following:

1. 57 out of 90 (63%) of the candidates were driver related and 22 out of 30 (73%) of the size checking cases, 9 out of 11 (82%) of the type selection cases and 19 out of 31 (61%) of the shallow copy cases occur in drivers.
2. 4 out of 5 (80%) of the double-fetch bugs we found inside drivers and belong to the size checking category.

Overall, this leads to the conclusion that most double fetches do not cause double-fetch bugs and that double fetches are more likely to occur in drivers. However, as soon as a double fetch is due to size checking, developers have to be careful: Four out of 22 size checking scenarios in drivers turned out to be double-fetch bugs.

4.2 Analysis of Three Open Source Kernels

Based on the double fetch basic pattern matching and manual analysis, we refined our double fetch pattern and developed a new double-fetch bug detection analysis based on the Coccinelle engine. In order to fully evaluate our approach, we analyzed three popular open source kernels, namely Linux, Android, and FreeBSD. Results are shown in Table 2.

For the Linux kernel, the experiment was conducted on version 4.5, which was the newest version when the experiment was conducted. The analysis took about 10 minutes and reported 53 candidate files. An investigation of the 53 candidates revealed five true double-fetch bugs, which were also found by the previous manual analysis. Among the reported files, 23 were size checking related, and 6 were type selection related.

For Android, even though it uses Linux as its kernel as well, we analyzed version 6.0.1 which is based on Linux 3.18. There are still differences between the Android kernel and original Linux kernel: A kernel for Android is a mainstream Linux kernel, with additional drivers for the specific Android device, and other additional functionality, such as enhanced power management or faster graphics support. Our analysis took about 9 minutes and reported 48 candidate files, including seven files that were not included in the original Linux kernel reports. Among the reported candidates, three were true double-fetch bugs, including two that were shared with the Linux 4.5 report above, and one that was only reported for Android. Among the results, 18 candidates were size checking related, and six candidates were type selection related.

For FreeBSD, we needed to change the transfer functions `copy_from_user()` and `__copy_from_user()` to the corresponding ones in FreeBSD, `copyin()` and `copyin_nofault()`. We obtained the source code from the master branch[3]. This analysis took about 2 minutes and only 16 files were reported, but none of them turned out to be a vulnerable double-fetch bug. Among the reported candidates, eight were size checking related, and three were type selection related. It is interesting to note that 5 out of these 16 files were benign double fetches, which would have been double-fetch bugs but were prevented by additional checking schemes. The developers of FreeBSD seem to be more aware of double-fetch bugs and try to actively prevent them. In comparison, for Linux, only 5 out of the 53 reports were protected by additional checking schemes.

In this experiment, we only counted the size checking and type selection cases because the refined pattern matching approach discards shallow copy cases that are not able to cause a double-fetch bug. Our approach matches the double fetch pattern that fetches data from the same memory region, which ignores the first buffer fetches in the case of a shallow copy and only considers multiple fetches to the same second buffer. Such shallow copy cases usually combine with other scenarios such as size checking and type selection. In Table 2, the size checking cases of the Linux kernel also includes one case that occurred in a shallow copy scenario.

5 Discussion

In this section, we discuss the discovered double-fetch bugs and vulnerabilities in Linux 4.5 and how double-fetch bugs can be prevented in the presence of double-fetch situations. We also interpret our findings and the limitations of our approach.

5.1 Detected Bugs and Vulnerabilities

Based on our approach, we found six double-fetch bugs in total. Five of them are previously unknown bugs that have not been reported before (CVE-2016-5728, -6130, -6136, -6156, -6480), and the sixth one (CVE-2015-1420) is a double-fetch bug present in the newest Android (version 6.0.1) which is based on an older Linux kernel (version 3.18) containing the bug, which has been fixed in the mainline Linux kernel since Linux 4.1. Three of the five new bugs are exploitable double-fetch vulnerabilities (CVE-2016-5728, -6136, -6480). Four of the five are in drivers (CVE-2016-5728, -6130, -6156, -6480). All bugs have been reported to the Linux kernel maintainers who have confirmed them. All of these reported bugs are fixed as of Linux 4.8. We did not find any new double-fetch bugs in FreeBSD. Details on the detected bugs are shown in Table 3.

The presented approach identifies a large number of double-fetch situations for which only a small number are double-fetch bugs (or even vulnerabilities). However, even though the cases we call benign double-fetch situations are not currently faulty, they could easily turn into a double-fetch bug or vulnerability when the code is updated without paying special attention to the double-fetch situation. We observed an occurrence of such a situation when investigating the patch history of the double-fetch bug CVE-2016-5728. A reuse of the second fetched value was introduced when the developer moved functionality from the MIC host driver into the Virtio Over PCIe (VOP) driver, therefore introducing a double-fetch bug. A major part of our future work will be preventing such benign double fetch situations from turning into harmful ones.

We did not find any false negatives while manually checking random samples of Linux kernel source code files.

5.2 Comparison

Only a few systematic studies have been conducted on double fetches. Bochspwn [14, 13] is the only approach similar enough to warrant a comparison with. An analysis of Linux 3.5.0 with Bochspwn did not find any double-fetch bug, while producing up to 200KB of double fetch logs. In the same kernel, our approach identi-

[3] From GitHub as of July 2016 (https://github.com/freebsd/freebsd)

Table 3: Description of Identified Double Fetch Bugs and Vulnerabilities (*)

IDs	File	Description
CVE-2016-5728*	`mic_virtio.c` MIC architecture VOP (Virtual I/O Over PCIe) driver *Linux 4.5*	Race condition in the `vop_ioctl` function allows local users to obtain sensitive information from kernel memory or cause a denial of service (memory corruption and system crash) by changing a certain header, aka a “double fetch” vulnerability. *Belongs to the size checking scenario.*
CVE-2016-6130	`sclp_ctl.c` IBM S/390 SCLP console driver *Linux 4.5*	Race condition in the `sclp_ctl_ioctl_sccb` function allows local users to obtain sensitive information from kernel memory by changing a certain length value, aka a “double fetch” vulnerability. *Belongs to the size checking scenario.*
CVE-2016-6136*	`auditsc.c` Linux auditing subsystem *Linux 4.5*	Race condition in the `audit_log_single_ execve_arg` function allows local users to bypass intended character-set restrictions or disrupt system-call auditing by changing a certain string, aka a “double fetch” vulnerability.
CVE-2016-6156	`cros_ec_dev.c` Chrome OS Embedded Controller driver *Linux 4.5*	Race condition in the `ec_device_ioctl_xcmd` function allows local users to cause a denial of service (out-of-bounds array access) by changing a certain size value, aka a “double fetch” vulnerability. *Belongs to the size checking scenario.*
CVE-2016-6480*	`commctrl.c` Adaptec RAID controller driver *Linux 4.5*	Race condition in the `ioctl_send_fib` function allows local users to cause a denial of service (out-of-bounds access or system crash) by changing a certain size value, aka a “double fetch” vulnerability. *Belongs to the size checking scenario.*
*CVE-2015-1420**	`fhandle.c` File System *Android 6.0.1, (Linux 3.18)*	Race condition in the `handle_to_path` function allows local users to cause a denial of service (out-of-bounds array access) by changing a certain size value, aka a “double fetch” vulnerability. *Belongs to the size checking scenario.*

fied 3 out of the above discussed 6 double-fetch bugs (the other 3 bugs we found are in files that were not present in Linux 3.5.0).

It is likely that Bochspwn could not find these bugs because they were present in drivers. Indeed, dynamic approaches cannot support drivers without corresponding hardware or simulations of hardware. Bochspwn reported an instruction coverage of only 28% for the kernel, while our approach statically analyses the complete source code.

As for efficiency, our approach takes only a few minutes to conduct a path-sensitive exploration of the source code of the whole Linux kernel. In contrast, Bochspwn introduces a severe runtime overhead. For instance, their simulator needs 15 hours to boot the Windows kernel.

While it only took a few days to investigate the 90 double-fetch situations, Jurczyk and Coldwind did not report the time they needed to investigate the 200KB of double fetch logs generated by their simulator.

5.3 Double-Fetch Bug Prevention

Even though we provide an analysis to detect double-fetch bugs, developers must still be aware of how they occur and preemptively prevent double-fetch bugs. Human mistakes are to be expected in driver development when dealing with variable messages leading to new double-fetch situations.

(1) Don’t Copy the Header Twice. Double-fetch situations 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. For example, the double-fetch vulnerability in Android 6.0.1 (Linux 3.18) is resolved in Linux 4.1 by only copying the body in the second fetch.

(2) Use the Same Value. A double-fetch situation turns into a bug when there is a use of the “same” data from both fetch operations because a (malicious) user can change the data between the two fetches. If developers only use the data from one of the fetches, problems are avoided. According to our investigation, most of the double-fetch situations are benign because they only use the first fetched value.

(3) Overwrite Data. There are also situations in which the data has 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 and eliminate the double-fetch bug is to overwrite the header from the second fetch with the header that has been fetched first. Even if a malicious user changed the header between the two fetches, the change would have no impact. This approach is widely adopted in FreeBSD code, such as in `sys/dev/aac/aac.c` and `sys/dev/aacraid/aacraid.c`.

(4) Compare Data. Another way to resolve a double-fetch bug is to compare the data from the first fetch to the data of the second fetch before using it. If the data is not the same, the operation must be aborted.

(5) Synchronize Fetches. The last way to prevent a double-fetch bug is to use synchronization approaches to guarantee the atomicity of two inseparable operations, such as locks or critical sections. As long as we guarantee that the fetched value cannot be changed between the two fetches, then nothing wrong will come out of fetching multiple times. However, this approach will incur performance penalties for the kernel, as synchronization is introduced on a critical section.

Since the *Compare Data* approach does not need to modify very much of the source code, most of the identified double-fetch bugs we found have been patched in this way by the Linux developers (CVE-2016-5728, -6130, -6156, -6480). If the overlapped data sections from the two fetches are not the same, the kernel will now return an error. One can argue that it would have been better to avoid the double fetch of the headers with any of the other first three recommendations. However, comparing the data has two advantages: it not only allow detecting attacks by malicious users but also protects from situation in which the data is changed without malicious intent (e.g., by some bug in user space code).

We have implemented the *Compare Data* approach in Coccinelle as an automatic patch that injects code to compare the data from the first fetch with the data from the second fetch at places where a double-fetch bug has been found. It is able to automatically patch all size checking double-fetch bugs, which accounts for most of the identified bugs (5/6).

5.4 Interpretation of Results

Double fetches are a fundamental problem for kernel development. Popular operating systems like Windows, Linux, Android, and FreeBSD all had double-fetch bugs and vulnerabilities in the past. Double-fetch issues have a long history, and one bug we identified (CVE-2016-6480) has existed for over ten years.

Double fetches are prevalent and sometimes inevitable in kernels. We categorized three typical double fetch scenarios from the occurrences we detected. 63% of these double fetches occur in drivers, which implies that drivers are the hard-hit area. Four out of the five new bugs belong to size checking scenarios, indicating that variable length message processing needs vetting for double-fetch bugs.

In the Linux kernel, double-fetch bugs are more complex than in Windows because transfer functions separate the fetches from the uses in a double-fetch bug, making it harder to separate benign from vulnerable double fetches. A previous dynamic approach has not found any double-fetch bug in Linux, where our static approach found some, demonstrating the power of a simple static analysis.

Our approach requires manual inspection, however, the manual inspection does not have to be repeated for the full kernel as future analyses can be limited to changed files. Moreover, developing a static analysis that automatically identifies double-fetch bugs with higher accuracy would have cost much more time than developing our current approach, running it on different kernels, and the manual investigating the results together. Also, before our analysis and categorization, it was not known in which situations double-fetch bugs occur in the Linux kernel—knowledge that was needed in order to design a more precise static double-fetch bug analysis. With the refined approach, one would only have had to look at the 53 potential double-fetch bugs, not at all 90 double-fetch situations. Therefore, the manual analysis part of our approach is inevitable but highly beneficial.

As for prevention, all of the four size checking bugs are patched by the *Compare Data* method, indicating the double fetches are not avoided completely as the patched situations still abort the client program by returning an error. Moreover, even benign double-fetch situations are not safe because they can turn into harmful ones easily. One such bug (CVE-2016-5728) was introduced from a benign double-fetch situation by a code update. However, most of these potential cases are not fixed as they are currently not vulnerable.

Even if a double fetch is benign, i.e., is not vulnerable, it can be considered a performance issue since one of the fetches (invocations of the transfer functions) is redundant.

5.5 Limitations

We focused on analyzing situations in which double fetches occur in Linux with a pattern-based analysis of the source code. However, the nature of the analysis prevents the detection of double fetches that occur on a lower level, e.g., in preprocessed or compiled code.

Double-fetch bugs can even occur in macros. In one such case [24], the macro fetches a pointer twice, the first time to test for `NULL` and the second time to use it. However, due to the potential pointer change between the two fetches, a null-pointer crash may be caused.

A double-fetch bug can also be introduced through compiler optimization. It then occurs in the compiled binary but not in the source code. Wilhelm [37] recently found such a compiler-generated double-fetch bug in the Xen Hypervisor, which is because the pointers to shared memory regions are not labeled as *volatile*, allowing the compiler to turn a single memory access into multiple accesses at the binary level, since it assumes that the memory will not be changed.

6 Related Work

So far, research conducted on double-fetch analysis has exclusively focused on dynamic analysis, whereas we proposed a static analysis approach. In addition to the already discussed work on Bochspwn [14, 13], there are also a few related studies as follows.

Wilhelm [37] used a similar approach to Bochspwn to analyze memory access pattern of para-virtualized devices' backend components. His analysis identified 39 potential double fetch issues and discovered three novel security vulnerabilities in security-critical backend components. One of the discovered vulnerabilities does not exist in the source code but is introduced through compiler optimization (see the discussion in Section 5.5). Moreover, another discovered vulnerability in the source code is usually not exploitable because the compiler optimizes the code in a way that the second fetch is replaced with a reuse of the value of the first fetch.

Double-fetch race conditions are very similar to Time-Of-Check to Time-Of-Use (TOCTOU) race conditions caused by changes occurring between checking a condition and the use of the check's result (by which the condition no longer holds). The data inconsistency in TOCTOU is usually caused by a race condition that results from improper synchronized concurrent accesses to a shared object. There are varieties of shared objects in any computer system, such as files [2], sockets [36] and memory locations [39], therefore, a TOCTOU can exist in different layers throughout the system. TOCTOU race conditions often occur in file systems and numerous approaches [5, 9, 18, 4, 27] have been proposed to solve these problems, but there is still no general, secure way for applications to access file systems in a race-free way.

Watson [36] worked on exploiting wrapper concurrency vulnerabilities that come from system call interposition. He focused on the wrapper vulnerabilities that will lead to security issues such as privilege escalation and audit bypass. By identifying resources relevant to access control, audit, or other security functionality that are accessed concurrently across a trust boundary, he found vulnerabilities from the wrappers and demonstrated the exploit techniques with examples. He also categorized the Time-Of-Audit to Time-Of-Use and Time-Of-Replacement to Time-Of-Use issues in addition to the Time-Of-Check to Time-Of-Use issue. However, he focused on the system call interposition security extensions rather than the kernel as we do. He did not provide details of how he found these vulnerabilities either.

Yang et al. [39] cataloged concurrency attacks in the wild by studying 46 different types of exploits and presented their characteristics. They pointed out that the risk of concurrency attacks was proportional to the duration of the vulnerability window. Moreover, they found that previous TOCTOU detection and prevention techniques are too specific and cannot detect or prevent general concurrency attacks.

Coccinelle [17], the program matching and transformation engine we use in our approach, was initially targeted for collateral evolution in Linux drivers, but now is widely used for finding and fixing bugs in systems code. With Coccinelle, Nicolas et al. [26, 25] performed a study of all the versions of Linux released between 2003 and 2011, ten years after the work of Chou et al. [7], who gave the first thorough study on faults found in Linux. Nicolas et al. pointed out that the kind of faults considered ten years ago were still relevant, and were still present in both new and existing files. They also found that the rate of the considered kinds of faults were falling in the driver directory, which supported Chou et al.

7 Conclusion

This work provides the first (to the best of our knowledge) static analysis of double fetches in the Linux kernel. It is the first approach able to detect double-fetch vulnerabilities in the complete kernel including all drivers and all hardware architectures (which was impossible using dynamic approaches). Based on our pattern-based static analysis, we categorized three typical scenarios in which double fetches are prone to occur. We also provide recommended solutions, specific to typical double-fetch scenarios we found in our study, to prevent double-fetch bugs and vulnerabilities. One solution is used to automatically patch double-fetch bugs, which is able to automatically patch all discovered bugs occurring in the size-checking scenario.

Where a known dynamic analysis of the Linux, FreeBSD, and OpenBSD kernels found no double-fetch bug, our static analysis discovered six real double-fetch bugs, five of which are previously unknown bugs, and three of which are exploitable double-fetch vulnerabilities. All of the reported bugs have been confirmed and fixed by the maintainers. Our approach has been adopted by the Coccinelle team and is currently being integrated into the Linux kernel patch vetting.

Acknowledgments

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 The National Key Research and Development Program of China (2016YFB0200401), by the program for New Century Excellent Talents in University, by the National Science Foundation (NSF) China 61402492, 61402486, 61379146, 61472437,and by the laboratory pre-research fund (9140C810106150C81001).

References

[1] Bug 166248 – CAN-2005-2490 sendmsg compat stack overflow. `https://bugzilla.redhat.com/show_bug.cgi?id=166248`.

[2] Bishop, M., Dilger, M., et al. Checking for race conditions in file accesses. *Computing systems 2*, 2 (1996), 131–152.

[3] Brunel, J., Doligez, D., Hansen, R. R., Lawall, J. L., and Muller, G. A foundation for flow-based program matching: Using temporal logic and model checking. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2009).

[4] Cai, X., Gui, Y., and Johnson, R. Exploiting UNIX file-system races via algorithmic complexity attacks. In *30th IEEE Symposium on Security and Privacy* (2009), pp. 27–41.

[5] Chen, H., and Wagner, D. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security* (2002), pp. 235–244.

[6] Chen, J., and MacDonald, S. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging* (2008), p. 8.

[7] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)* (2001).

[8] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers*. O'Reilly Media, Inc., 2005.

[9] Cowan, C., Beattie, S., Wright, C., and Kroah-Hartman, G. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *USENIX Security Symposium* (2001), pp. 165–176.

[10] Engler, D., and Ashcraft, K. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)* (2003), ACM, pp. 237–252.

[11] Hammou, S. Exploiting Windows drivers: Double-fetch race condition vulnerability, 2016. `http://resources.infosecinstitute.com/exploiting-windows-drivers-double-fetch-race-condition-vulnerability/`.

[12] Huang, J., and Zhang, C. Persuasive prediction of concurrency access anomalies. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (2011), pp. 144–154.

[13] Jurczyk, M., and 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`.

[14] Jurczyk, M., and Coldwind, G. Identifying and exploiting windows kernel race conditions via memory access patterns. Tech. rep., Google Research, 2013. `http://research.google.com/pubs/archive/42189.pdf`.

[15] Kasikci, B., Zamfir, C., and Candea, G. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012), pp. 185–198.

[16] Kasikci, B., Zamfir, C., and Candea, G. RaceMob: crowdsourced data race detection. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles* (2013), pp. 406–422.

[17] Lawall, J., Laurie, B., Hansen, R. R., Palix, N., and Muller, G. Finding error handling bugs in OpenSSL using Coccinelle. In *European Dependable Computing Conference (EDCC)* (2010), pp. 191–196.

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

[19] Lu, K., Wu, Z., Wang, X., Chen, C., and Zhou, X. RaceChecker: efficient identification of harmful data races. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2015), pp. 78–85.

[20] Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008), pp. 329–339.

[21] Lu, S., Park, S., and Zhou, Y. Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering 38*, 4 (2012), 844–860.

[22] Lu, S., Tucek, J., Qin, F., and Zhou, Y. AVIO: detecting atomicity violations via access interleaving invariants. In *ACM SIGARCH Computer Architecture News* (2006), vol. 34, pp. 37–48.

[23] Lucia, B., Ceze, L., and Strauss, K. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. *ACM SIGARCH computer architecture news 38*, 3 (2010), 222–233.

[24] McKenney, P. E. list: Fix double fetch of pointer in hlist_entry_safe(), 2013. `https://lists.linuxfoundation.org/pipermail/containers/2013-March/031996.html`.

[25] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., and Muller, G. Faults in Linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[26] Palix, N., Thomas, G., Saha, S., Calves, C., Muller, G., and Lawall, J. Faults in Linux 2.6. *ACM Transactions on Computer Systems (TOCS) 32*, 2 (2014), 4.

[27] Payer, M., and Gross, T. R. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (2012), pp. 215–226.

[28] Pratikakis, P., Foster, J. S., and Hicks, M. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS) 33*, 1 (2011), 3.

[29] Ryzhyk, L., Chubb, P., Kuz, I., and Heiser, G. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), pp. 275–288.

[30] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS) 15*, 4 (1997), 391–411.

[31] Sen, K. Race directed random testing of concurrent programs. *ACM SIGPLAN Notices 43*, 6 (2008), 11–21.

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

[33] Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W., and Zheng, W. Do i use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 160–174.

[34] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst. 23*, 1 (Feb. 2005), 77–110.

[35] Voung, J. W., Jhala, R., and Lerner, S. RELAY: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2007), pp. 205–214.

[36] Watson, R. N. Exploiting concurrency vulnerabilities in system call wrappers. In *First USENIX Workshop on Offensive Technologies (WOOT)* (2007).

[37] Wilhelm, F. Tracing privileged memory accesses to discover software vulnerabilities. Master's thesis, Karlsruher Institut für Technologie, 2015.

[38] Wu, Z., Lu, K., Wang, X., and Zhou, X. Collaborative technique for concurrency bug detection. *International Journal of Parallel Programming 43*, 2 (2015), 260–285.

[39] Yang, J., Cui, A., Stolfo, S., and Sethumadhavan, S. Concurrency attacks. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism* (2012).

[40] Zhang, M., Wu, Y., Lu, S., Qi, S., Ren, J., and Zheng, W. Ai: a lightweight system for tolerating concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 330–340.

[41] Zhang, W., Sun, C., and Lu, S. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)* (2010), pp. 179–192.