DFTinker: Automated Detecting and Fixing Double-fetch Bugs with Transactional Memory

Yingqi Luo[1], Pengfei Wang[1], Xu Zhou[1], and Kai Lu[1]

National University of Defense Technology, Changsha Hunan, P.R.China

Abstract. The double-fetch bug is a situation where the operating system kernel fetches the supposedly same data twice from the user space, whereas the data is unexpectedly changed by the user thread. It could cause fatal errors such as kernel crashes, information leakage, and privilege escalation. Previous research focuses on the detection of double-fetch bugs, however, the fix of such bugs still relies on manual efforts, which is inefficient. This paper proposes a comprehensive approach to automatically detect and fix double-fetch bugs. It uses a static pattern-matching method to detect double-fetch bugs and automatically fix them with the support of the transactional memory (Intel TSX). A prototype tool named DFTinker is implemented and evaluated with prevalent kernels. Compared with prior works, DFTinker can automatically detect and fix double-fetch bugs at the same time. It owns a high code coverage, accuracy, and the performance overhead is only 1.3%.

1 Introduction

The wide use of multi-core hardware is making concurrent programs increasingly pervasive, especially in operating systems, network systems, and even IoT devices. However, the reliability of such system is severely threatened by the notorious concurrency bugs, such as errors caused by the race condition, which are difficult to detect owing to the uncertainty introduced by the thread schedule. Among all the concurrency bugs, the double-fetch bug is one of the most special and significant types.

For the sake of security, the memory model in modern operating systems isolates the kernel space and the user space. User applications use kernel functionality by the invocation of syscalls. Thus, kernel functions inevitably need to use data from the user space, which is untrustable because the data can be tampered from the user side. A double-fetch bug happens when the kernel fetches the supposedly same data twice from the user space, whereas the data is unexpectedly changed by a concurrently running user thread under a race condition. It could cause fatal errors such as kernel crashes, information leakage, and privilege escalation [15].

Previous research focuses on the detection of double-fetch bugs. Dynamic approaches [7,8,16] detect double-fetch bugs by tracing memory accesses. However, such approaches are limited by the path coverage. They 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. Static approaches detect double-fetch bugs based on the identification of transfer functions [14,17], however, the accuracy and efficiency of such approaches are undesirable as they lack runtime information and still rely on manual efforts to confirm the bug. In addition, none of the previous works provides a practical solution on automatically fixing double-fetch bugs except some prevention suggestions. Thus, the fix of double-fetch bugs still relies on manually locating and rewriting the source code, and an automatic solution is in urgent need.

This paper proposes a comprehensive approach to automatically detect and fix double-fetch bugs. In the first phase, a static pattern-matching method based on the Coccinelle engine is used to identify double-fetch bugs. In the second phase, the identified bug is automatically fixed based on the support of the Intel Transactional Synchronization Extension (TSX) [18].In summary, the main contribution of this paper is as follows:

- This paper proposes a comprehensive approach to automatically detect and fix double-fetch bugs at one time. The approach can cover all architectures in one detect execution, need no manual involvement, and achieve a more accurate result than previous research.
- A prototype tool named DFTinker is implemented. We have made it publicly available, hoping it can be useful for future study.
- DFTinker is evaluated with prevalent real kernels. Results show that it is effective and efficient in automatically detecting and fixing double-fetch bugs. The performance overhead of the fixed program is only 1.3% on average.

The rest of paper is organized as follows. Section 2 introduces background about double-fetch bugs, the Coccinelle engine, and Intel TSX. Section 3 presents the design details of our approach. Section 4 shows the implementation of DFTinker. Section 5 presents the evaluation of DFTinker.Section 6 discusses this work, followed by the related work in Section 7.The conclusion is in Section 8.

2 Background

2.1 The Double-fetch Bug

In modern operating systems, kernel space is always separated from user space for safety [13]. Kernel code run in kernel space and if there is a need to get data from users, it will use specific functions, termed *transfer functions*. In Linux kernel, there are four typical transfer functions, `get_user()`, `put_user()`, `copy_from_user()`, `copy_to_user()`. All their effects are fetching data or transferring data between the kernel space and the user space. However, there are many cases where kernel fetches data from the same user address twice or more times. The first time kernel fetches data from the user space, it may check whether the data is legal or not. If it is a legal data, the kernel will conduct the second fetch to get the whole data into the kernel. Malicious changes between two fetches may cause kernel get unexpected data at second fetch, leading to system crashes or even worse results.

```
55  static int sclp_ctl_ioctl_sccb(void __user *user_area)
56  {
          ...
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  }
```

Fig. 1. A Double-Fetch Bug (CVE-2016-6130) in File **/drivers/s390/char/sclp_ctl.c** of Linux Kernel 4.5

Fig.1 shows CVE-2016-6130, a Linux kernel double-fetch bug in the file `/drivers/s390/char/sclp_ctl.c`. In this case, the first fetch occurs in line 68, it copies the data pointed by `ctl_sccb.sccb` from user space to the kernel space. Then, it checks the validity of `sccb->length` at line 72. Finally, it fetches the data again with the checked parameter `sccb->length` at line 74. Thus, when `sccb->length` is changed to a very large value between the two fetches, kernel information can be leaked to the user space at line 81.

Wang *et al.* [14] summarized three scenarios of double-fetch bugs in his paper. They are as follows:

Type Selection. Type selection is the scenario where the data that kernel fetches the first time from user space is a message header, and it is used to identify the message type, thus, the kernel can handle different types of messages. According to prior work, it's common in Linux drivers to use a `switch` statement to handle multiple types of messages in one function. If the messages were changed maliciously after the first fetch, it may cause the kernel handle with the unexpected data, which would result in buffer overflow or even worse situations.

Size Checking. Size checking is the scenario where the kernel fetches the messages' length at the first fetch, after checking the validity of the length and allocating the right space for the message, the kernel gets the message to the kernel space at the second fetch. This scenario is also vulnerable. Once the message length is replaced by a larger value, a buffer overflow may occur, for a larger string copying into a limited space. CVE-2016-5728, CVE-2016-6130, CVE-2016-6156, CVE-2016-6480, CVE-2015-1420 all belong to this scenario.

Shallow Copy. The last scenario is shallow copy where the data copied from user space to kernel space contains a pointer to another buffer in user space. In this scene, the second buffer in user space needs another transfer function, i.e.,

the second fetch. Fortunately, this kind of double fetches may not harm for each fetch transferring data from different space.

However, these three types cannot cover all double-fetch bugs and own a high false alarm rate, we propose a better model in section 3.1.

2.2 Coccinelle Engine

Coccinelle [6,12] engine is a program matching and transformation engine. It uses language SmPL (Semantic Patch Language) as rules to perform matching and transformations in C code. Coccinelle was initially targeted towards performing collateral evolutions in Linux, and it is widely used for finding and fixing bugs in system code now [10].

One of the advantages of Coccinelle engine is path-sensitive [6], it is specially optimized to achieve a better performance at traversing paths. Besides, Coccinelle engine will ignore spaces, newlines, and comments, which reduces difficulties of developers' programming. Coccinelle will not expand macros, so macro operations, such as `__get_user()`, can be directly used in pattern rules.

2.3 Transactional Memory and Intel TSX

Traditionally, transactional memory [3–5, 9] is used to simplify concurrent programming. It allows executing load and store instructions in an atomic way. Transactional memory systems provide high-level instructions to developers so as to avoid low-level coding, and this achieves a better access model to shared memory in concurrent programming.Hardware transactional memory achieves transactions by processors, caches, and bus protocol. It provides opportunities to implement dynamic schedules according to specific CPU instructions.

Intel Transactional Synchronization Extensions (TSX) came out in 2013 [1, 2, 18], making hardware transactional memory available in commodity processors. It is an extension to the x86 instruction set architecture (ISA), providing hardware transactional memory support.

Intel TSX provides two interfaces for transactional execution, Hardware Lock Elision(HLE) and Restricted Transactional Memory(RTM).

HLE. HLE provides two new prefixes of instruction, `XACQUIRE` and `XRELEASE`. They share the same opcode of `REPNE` and `REPE`. Once the processor does not support TSX, prefixes `REPNE` and `REPE` will be ignored and it does not affect the instruction execution. Thus HLE can be backward compatible. If there is a conflict, transaction will rerun from the `XACQUIRE`-prefixed instruction.

RTM. RTM provides more friendly usability for developers. It defines three new instructions: `XBEGIN`, `XEND`, and `XABORT`. Programmers can use `XBEGIN` and `XEND` to specify the code region needs to be transactional executed. `XABORT` is used to abort a hardware transaction. Moreover, a `XTEST` instruction can be used to tell whether the processor is in transactional execution mode. As RTM cannot guarantee that the transactions been executed successfully, programmers need to prepare a fallback path in case never successes.

3 Design

3.1 Detection of Double-Fetch Bugs

According to section 2.1, we can figure out that double-fetch bugs and transfer functions are related closely, for each fetch means an invocation of a transfer function. However, it may also not be a double-fetch bug if there are two transfer functions. It has to meet the condition that the double fetches get the data from the same user address at least. Since there are many complex situations in kernel code, such as assignment and pointer, it needs further and thorough analysis.

Wang *et al.* used four transfer functions in his study, they are `get_user()`, `__get_user()`, `copy_from_user()`, and `__copy_from_user()`, whose functionality is transferring data from user space to kernel space. Besides, he proposed six rules to improve precision and find corner cases, as Fig.2 shows, they are as follows:

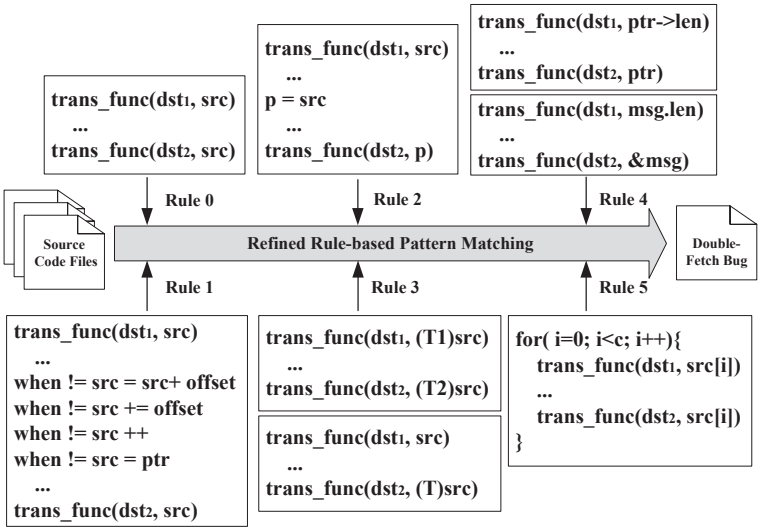

Fig. 2. Refined Coccinelle-Based Double-Fetch Bugs Detection [14]

Rule 0: Basic pattern matching rule. This is the basic scene when two fetches get the data from the exact same user address. As to the more complex situations like assignment and pointer, the other five rules are used.

Rule 1: No pointer change. This rule is used to eliminate the cases when the pointer to the user space changes between two fetches, such as self-increment, adding or subtracting an offset. This rule can reduce the false positive rate of detection.

Rule 2: Pointer aliasing. There are many pointer assignments in kernel code, so different pointers in double fetches may point to the same user address.

```
96   static struct autofs_dev_ioctl * copy_dev_ioctl(struct autofs_dev_ioctl __user *in)
97   {
98    struct autofs_dev_ioctl tmp, *res;
99
100   if (copy_from_user(&tmp, in, AUTOFS_DEV_IOCTL_SIZE))
101     return ERR_PTR(-EFAULT);
102
103   if (tmp.size < AUTOFS_DEV_IOCTL_SIZE)
104     return ERR_PTR(-EINVAL);
105
106   if (tmp.size > AUTOFS_DEV_IOCTL_SIZE + PATH_MAX)
107     return ERR_PTR(-ENAMETOOLONG);
108
109   res = memdup_user(in, tmp.size);
110   if (!IS_ERR(res))
111     res->size = tmp.size;
        ...
114  }
```

Fig. 3. An Undetectable Double-Fetch Bug in File **/fs/autofs4/dev-ioctl.c**

Wang *et al.* used an assignment between double fetches to detect this situation as shown in Fig.2.

Rule 3: Explicit type conversion. Pointer type conversion is always used while fetching data from the user space. At the first fetch, the pointer may be converted to a message header pointer and converted to a whole message pointer at the second fetch. This rule can reduce the false negative rate of detection.

Rule 4: Combination of element fetch and pointer fetch. Another complex situation is that the user addresses fetched twice are not exactly the same. For example, at the first fetch, the kernel fetches a data member of a struct such as `ptr->length`, and then, after checking validity or preparation, the kernel fetches the whole struct using `ptr` at the second fetch.

Rule 5: Loop involvement. The last rule is about loop operations. As mentioned above, Coccinelle engine is path-sensitive, so it will expand a loop multiple times, i.e., each transfer function will be scanned more than one time. Thus, the transfer function at the end of a loop and transfer function at the beginning of the next loop will be paired as a double fetch, which should be excluded as false positive.

However, these rules are not strong enough to cover all double-fetch bugs. Such as function `copy_dev_ioctl()` in file `/fs/autofs4/dev-ioctl.c`, as shown in Fig.3. In this double-fetch bug, the first fetch happens at line 100 with a transfer function `copy_from_user()` and the second fetch happens at line 109 with a normal function `memdup_user()`. They both copy the data from the same user address pointed by the pointer `in` to the kernel space. However, function `memdup_user` is not one of the four transfer functions Wang *et al.* used, so it won't be detected by Wang *et al.*'s method. But in fact, function `memdup_user` indeed contains a transfer function `copy_from_user()` and some other operations, it can be regarded as a transfer function completely in this case.

The rules improved are as follows:

Table 1. Expanded Transfer Functions

No.	Name	Type	Parameter
1	get_user	macro	dst, src
2	__get_user	macro	dst, src
3	unsafe_get_user	macro	dst, src, err
4	__copy_in_user	macro	des, src, len
5	__copy_user	function	dst, src, len
6	__copy_user_zeroing	function	dst, src, len
7	copy_from_user	function	dst, src, len
8	__copy_from_user	macro	dst, src, len
9	__copy_from_user_inatonmic	macro	dst, src, len
10	strncpy_from_user	function	dst, src, len
11	strndup_user	function	src, len
12	memdup_user	function	src, len
13	memdup_user_nul	function	src, len
14	getname_flags	function	src, flags
15	getname	function	src

Add more transfer functions. Wang *et al.* used only four transfer functions in his experiment, `get_user()`, `__get_user()`, `copy_from_user()`, and `__copy_from_user()`. However, there are also many normal functions containing transfer functions, and their targets are transferring data from user space to kernel space as well, such as `memdup_user()` mentioned above. In addition to funcions, there are also many macros playing the same role in the kernel, like `__copy_from_user_inatomic()`. In the double-fetch bugs detection scenario, these normal functions and macros can be regarded as transfer functions directly.

Table 1 shows the 15 functions (and macros) used in double-fetch bugs detection and their types and parameters. Except for functions used, there are still many functions in kernel meet the condition that transferring data from user space to kernel space, however, many of them have not been used in the code. So these functions are abandoned and lifted efficiency.

Fix incomplete rules. Rules which Wang *et al.* proposed are theoretically correct. However, in the implementation phase, they were achieved incompletely, which leaded false negatives. For instance, in `/fs/nfsd/nfs4recover.c`, as shown in Fig.4, the first fetch happens at line 730, fetching the data pointed by `&cmsg->cm_xid`, the second fetch happens at line 753, fetching the data pointed by `src`, and there is no assignment or statement between two fetches. According to Rule 2 in section 3.1, this is not a double-fetch bug. But in fact, this is indeed a double-fetch bug for its assignment happens at line 717, thus `cmsg` and `src` point to the same user address actually. This case is missed because of careless implementation. We fixed the rule and reduced the false negative rate.

Remove more non-double-fetch bugs. Wang *et al.* used his pattern rules find 90 candidates files in total, it's still a little heavy for technicians to check

```
714 static ssize_t cld_pipe_downcall(struct file *filp, const char __user  *src, size_t mlen)
715 {
716   struct cld_upcall *tmp, *cup;
717   struct cld_msg __user *cmsg = (struct cld_msg __user *)src;
718   uint32_t xid;
719   struct nfsd_net *nn = net_generic(file_inode(filp)->i_sb->s_fs_info, nfsd_net_id);
721   struct cld_net *cn = nn->cld_net;
        ...
730   if (copy_from_user(&xid, &cmsg->cm_xid, sizeof(xid)) != 0) {
731     dprintk("%s:␣error.", __func__);
732     return -EFAULT;
733   }
        ...
752
753   if (copy_from_user(&cup->cu_msg, src, mlen) != 0)
754     return -EFAULT;
        ...
758 }
```

Fig. 4. An Undetectable Double-Fetch Bug in File **/fs/nfsd/nfs4recover.c**.

manually. To lower false positive rate, more situations are added to remove those non-double-fetch bugs:

1. The procedure returns after the first fetch. As shown in Fig.5, there are two fetches at line 850 and 879, and the first fetch is in a `IF` statement. This case will be matched with prior rules apparently. However, there is a `RETURN` statement after the first fetch at line 857, that means, the second fetch will never be executed if the first fetch is executed. Thus, this is not a double-fetch bug actually. There are many cases like this in the kernel code.

2. Two fetches are in the different branches Another situation is when the two fetches are in the different branches, just like a `SWITCH` statement. For instance, function `hysdn_conf_write` in `\drivers\isdn\hysdn\hysdn_procconf.c` is matched with double-fetch bugs detection. But its two fetches are located in different branches, the first fetch is in the `IF` statement with condition `cnf->state == CONF_STATE_DETECT` and the second fetch is in the `IF` statement with condition `cnf->state == CONF_STATE_POF`. These two conditions can never be satisfied at the same time, so this situation is also a non-double-fetch bug situation.

Besides, based on the improvement mentioned above, we summed up the other two types of double-fetch bugs. They are as follows:

1. Validity Checking. Validity checking is the situation when the header of a message is fetched to verify the validity of the message. Different from size checking and type selection, this header is only used in the validity verifying, whereas in other cases, the header will be used in memory allocation or `SWITCH` statements.

2. Reacquisition. Another type is Reacquisition. In this scenario, there is no obvious relation between the two fetches. It simply fetches the data from the same user address twice.

```
826 int isdn_ppp_write(int min, struct file *file, const char __user *buf, int count)
827 {
          ...
844     if (lp->isdn_device < 0 || lp->isdn_channel < 0) {
          ...
850       if (copy_from_user(protobuf, buf, 4))
851         return -EFAULT;
          ...
857       return 0;
858     }
859
860     if ((dev->drv[lp->isdn_device]->flags & DRV_FLAG_RUNNING) &&
          ...
878       cpy_buf = skb_put(skb, count);
879       if (copy_from_user(cpy_buf, buf, count))
880       {
881         kfree_skb(skb);
882         return -EFAULT;
883       }
          ...
904 }
```

Fig. 5. An Example when Procedure Returns after the First Fetch.

3.2 Automated Patching with Intel TSX

Previous research only proposed suggestions on preventing double-fetch bugs [14, 17], such as don't copy the header twice, use the same value, overwrite the second fetched data, compare data from the two fetches, and synchronize the fetches. However, we need a practical solution to automatically fix the bug.

Transactional memory, as an emerging parallel programming paradigm, provides opportunities to facilitate the dynamic schedules via speculative execution. Instead of pessimistically locking the shared memory locations to prevent potential conflicting concurrent updates, transactional executions optimistically elide the lock and speculatively perform memory operations. Thus transactional memory becomes a promising solution that provides programmer-friendly usability without sacrificing performance.

DFTinker's fixing function is implemented using Intel's Restricted Transactional Memory (RTM) software interface. RTM defines three new instructions: `XBEGIN`, `XEND`, and `XABORT`. Programmers can use `XBEGIN` and `XEND` to specify the begin and end of a hardware transaction and use `XABORT` to explicitly abort a hardware transaction. Additionally, one can adopt the `XTEST` instruction to check whether the processor is executing a code region in transactional execution mode. Due to its best-effort nature, RTM never guarantees successful commit of hardware transactions, which necessitates a fallback path to ensure forward progress.

As Fig.6 shows, a standard mutex lock is employed as the fallback path. Our lock method firstly initiates a hardware transaction via `XBEGIN` and checks the state of the mutex immediately at the beginning of the transaction. The locked state of the mutex lock indicates that there is a transaction executing on the fallback path, and all the speculative executions must be canceled. If the mutex

```
1: shared variable:
2:     mutex
3:
4: local variable:
5:     retries
6:
7: procedure LOCK( )
8:     retries ← 0
9:     _xbegin( )
10:    if lock_is_free(lock) then
11:        return
12:    else
13:        _xabort( )
14:    end if
15:    retries++
16:    if retries < MAX_RETRIES then
17:        goto line 9
18:    else
19:        mutex.lock( ) a
20:    end if
21: end procedure
22:
23: procedure UNLOCK( )
24:    if _xtest( ) then
25:        _xend( )
26:    else
27:        mutex.unlock( )
28:    end if
29: end procedure
```

Fig. 6. Speculative lock & unlock with HTM

lock is free, our lock method just returns (without touching the mutex variable) and leaves the processor in the transactional execution mode thus that it could perform updates to latents speculatively with zero synchronization overhead. Note that this is only a basic fallback scheme in Fig.6 for simplicity, more thorough and powerful fallback path could be established through scanning the *abort status* register to get the detailed cause for the abort.

4 Implementation

DFTinker was implemented on a Linux laptop running Ubuntu 16.04 x64, with one Intel i7-7700HQ 2.6GHz processor, 8GB of memory, 250GB SSD. The Coccinelle version was 1.0.4 with Python support.

The experiment used Linux 4.14.10, OpenBSD 6.2, FreeBSD 11.1, Android 7.0.0 and Darwin 10.13.3, which were the relatively newer version when the experiment was conducted. To prove that DFTinker can find out bugs reported

Table 2. Statistical Results of Detection of Double-Fetch Bugs

Kernel	Version	Files	Size Checking	Type Selection	Validity Checking	Reacquisition	Total Bugs
Linux	4.14.10	45614	13	5	4	2	24
FreeBSD	11.1	38811	7	2	2	1	12
OpenBSD	6.2	29704	0	0	0	0	0
Android	7.0.0 (3.18)	30479	14	7	5	15	41
Darwin	10.13.3	49105	3	1	0	0	4

by prior works, the experiment was also conducted on Linux 4.5, the same version Wang *et al.* used to experiment on.

To evaluate the efficiency of the fixed operating system, another Ubuntu 14.04 x64 was implemented. We chose five vulnerable functions from the detection results and fixed them by DFTinker. By comparing the run time of the invocation of the vulnerable functions before fixing and after fixing, we could get the performance overhead of DFTinker. This operating system ran on a server with a 4-core Intel Core i7-6700 CPU (clocked at 3.40GHz, supporting RTM) with each core possessing a private 32KB L1 cache and a private 256KB L2 cache and a shared 8MB L3 cache. The memory was 32GB and the storage was a 250GB SSD.

5 Evaluation

This section will discuss DFTinker's ability to detect double fetches and the performance of the operating system fixed by DFTinker.

5.1 Detection of Double-Fetch Bugs

At first, DFTinker was run to confirm Wang *et al.*'s work [14] with the Linux kernel 4.5, the same version Wang *et al.* used to experiment. It successfully reached Wang *et al.*'s result. All five bugs reported in Linux kernel 4.5 were found (CVE-2016-5728, CVE-2016-6130, CVE-2016-6136, CVE-2016-6156, CVE-2016-6480), which proved that DFTinker is as good as Wang *et al.*'s work.

Then it was applied to five open source kernels: Linux 4.14.10, FreeBSD 11.1, OpenBSD 6.2, Android 7.0.0 (kernel version 3.18) and Darwin 10.13.3. These were the relatively newer version when the experiment was conducted. The statistical result is shown as Table 2 and parts [1] of the detailed results are shown in Table 3.

1. Linux. The Linux kernel used is version 4.14.10 which was released on Dec 29, 2017. In this version, five double-fetch bugs reported by Wang *et al.* had already been fixed, but it still found 24 cases in total.

[1] Due to the limitation of the page, the full results of the double-fetch bugs detection are available at https://github.com/luoyyqq

Table 3. Part of the Results of Double-fetch Bugs

No.	File	Function	the First Fetch	the Second Fetch	Type
1	bus.c	__nd_ioctl	942	1025	Type-selection
2	commctrl.c	ioctl_send_fib	82	119	Size-checking
3	megaraid_mm.c	mega_m_to_n	3443	3467	Validity-checking
...					
25	aac.c	aac_ioctl_sendfib()	2999	3007	Size-checking
26	aacraid.c	aac_ioctl_sendfib()	2763	2771	Size-checking
27	bcm2835_vcio.c	vcio_ioctl()	66	72	Size-checking
...					
37	signal_32.c	do_sigreturn	86	93	Validity-checking
38	tcp.c	do_tcp_getsockopt	2774	2823	Reacquisition
39	ft1000_debug.c	ft1000_ioctl	551	566	Size-checking
...					
78	dtrace.c	dtrace_dof_copyin	11602	11626	Size-checking
79	nfs_syscalls.c	fhopen	620	626	Size-checking
80	nfs_vfsops.c	nfs_vfs_mount	1834	1872	Type-selection
81	ucode.c	copyin_update	92	117	Size-checking

2. FreeBSD. FreeBSD is an open-source Unix-like operating system, which is the most widely used open-source BSD distribution. The FreeBSD version was 11.1, which was released in July 2017. Note that FreeBSD is a little different from Linux, for FreeBSD uses `copyin()` and `copyin_nofault()` as transfer functions. Thus it needs to modify corresponding pattern code in DFTinker before the experiment. In FreeBSD, 12 cases were found in total.

3. OpenBSD. OpenBSD is also an open-source UNIX-like operating system and it is one of the three popular distributions of BSD. The OpenBSD version tested is 6.2, which was released in Oct 2017. In our experiment, it didn't find any double-fetch bug in OpenBSD. In fact, OpenBSD provides many security features in the system which are optional or unavailable in other operating systems, and developers audit source code for security frequently. This may be the reason why DFTinker can't find any double-fetch bug in OpenBSD.

4. Android. Android is a special distribution of Linux. It uses Linux kernel with specific modification. The Android version tested was 7.0.0 based on Linux kernel version 3.18. We found 41 double-fetch bugs in the Android kernel source.

5.Darwin. Darwin is a Unix operating system which is open-sourced by Apple Inc. in 2000. In our experiments, we chose Darwin version 10.13.3, which is the newest version when we conducted the experiments. We totally found 4 double-fetch bugs in Darwin.

5.2 Automated Patching with Intel TSX

As Coccinelle engine is accurate in locating lines of double-fetch bugs in the code, it is easy to fix `LOCK()` and `UNLOCK` operations to the code. According to

Table 4. Results of the Performance Tests of the Fixed Code

Vulnerable Functions	Files	Origin Runtime (µs)	Fixed Runtime (µs)	Average Overhead
ioctl_file_dedupe_range()	ioctl.c	556.25	563.25	1.3%
ll_dir_ioctl()	dir.c	438.72	444.12	1.2%
copy_dev_ioctl()	dev-ioctl.c	659.43	668.86	1.4%
_ctl_ioctl_main()	mpt3sas_ctl.c	445.12	450.11	1.1%
sg_scsi_ioctl()	scsi_ioctl.c	515.12	522.02	1.3%

the feature of transaction memory, all operations between `LOCK()` and `UNLOCK()` will be executed in a transaction, execution results will be committed if there are no conflicts. In other words, if there is a malicious user changes the data in the user space after the first fetch, all operations in the transaction will be aborted and rerun from `LOCK()`, which guarantees the consistency of the data. In our experiments, we fixed target functions using DFTinker and modified the data between two fetches. The results showed that the data fetched at the second time was same as the first time, which proved that DFTinker was effective in protecting double-fetch bugs.

To evaluate the overhead performance of the fixed code, an Ubuntu 14.04 x64 was used, whose kernel version was 4.6.1. DFTinker was used to fix the functions in the Table 4. Then, a program ran to call the target function for a thousand times and recorded its runtime. At last, it was compared the run time that ran on the operating system not been fixed. After testing for twenty times and taking the average, the fixed operating system owned a decent overhead performance of 1.3 %. The detailed results were shown in Table 4.

6 Discussion

Double-fetch bugs are significant problems in the operating systems, and they have been found in almost all the modern kernels, such as the Windows, Linux, FreeBSD, and Android [14]. Double-fetch bugs have a long history, and some of them existed over 10 years (CVE-2016-6480). Besides, potential double-fetch bugs (currently not buggy) can turn into buggy ones when the code is updated without paying special attention to the double fetch issue [14], such as CVE-2016-5728 and CVE-2016-6516. Thus, in this paper, we take all the potential double-fetch bug situations into consideration to conduct a thorough detection and fix.

Although DFTinker achieves a decent performance in detecting and fixing double-fetch bugs, it has limitations. It relies on the availability of the source code, which is suitable for in-house testing. It cannot handle the special case that the double-fetch bug does not reside in the source code but is introduced by the compiler (CVE-2015-8550) [16]. We will include such situations in the future work.

7 Related Work

Jurczyk and Coldwind [7,8] used a dynamic approach in their Bochspwn project to study double-fetch bugs in Windows. By tracing memory accesses, they successfully found double-fetch bugs in the Windows kernel. Wilhelm [16] 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. One of the discovered vulnerabilities does not exist in the source code but is introduced through compiler optimization. Inherently, such dynamic approaches have a low code coverage, i.e., code under strict conditions may never be tested. Furthermore, code that is being tested is limited to the emulation ability, thus double-fetch bugs in hardware devices that it cannot emulate may be missed. Our static approach has a better code coverage and can detect double-fetch bugs in the drivers, where the dynamic approaches are incapable of.

Wang *et al.* [14] is the first to systematically study double-fetch bugs in the Linux kernel. Based on the transfer functions, they use a static pattern-matching approach to detect double-fetch bugs. In addition to the identification of six real double-fetch bugs, they also propose five solutions to prevent double-fetch bugs. However, the accuracy of their detection approach is undesirable and they could not automatically fix the bug after identifying them. Our approach can automatically detect and fix double-fetch bugs at the same time.

Schwarz *et al.* [11] proposed a method using cache-attack and kernel-fuzzing techniques to detect, exploit, and eliminate double-fetch bugs in Linux syscalls. However, their approach is limited to Linux syscalls, whereas large numbers of the double-fetch bugs occur in non-syscall functions, such as functions in drivers, are missed. Thus, their approach suffers from a low code coverage, whereas our approach is free from that.

Xu *et al.* [17] proposed a formal definition of double-fetch bugs and used a static analysis based on LLVM IR and symbolic execution to detect such bugs. However, their definition takes all the potential situations into consideration, which are not currently buggy but only have the potential to turn into bugs when the code is updated. Besides, their approach needs to compile the source code to LLVM IR and specify the target architecture, thus, it detects only one architecture at one time, leading to the miss of bugs such as CVE-2016-6130. Our approach has a better code coverage, which can analyze the source code of all the architecture at one time. We also provide automatical fix after the bug is identified, which is efficient than previous works.

8 Conclusion

This paper proposes an approach to automatically detect and fix double-fetch bugs. We implement a prototype named DFTinker and evaluate it with real kernels. Experiments show that DFTinker is effective and efficient in automatically

detecting and fixing double-fetch bugs. DFTinker detected 81 cases in prevalent kernels and the performance overhead of the fixed program is only 1.3% on average.

References

1. B. Goel, R. Titos-Gil, A. Negi, S. A. Mckee, and P. Stenstrom. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *Parallel and Distributed Processing Symposium, 2014 IEEE International*, pages 615–624, 2014.
2. F. Haas and S. Metzlaff. Enhancing real-time behaviour of parallel applications using intel tsx. *Entomological Science*, 11(1):123126, 2014.
3. L. Hammond, V. Wong, M. Chen, and B. D. Carlstrom. Transactional memory coherence and consistency. In *International Symposium on Computer Architecture, 2004. Proceedings*, pages 102–113, 2004.
4. T. Harris, J. Larus, and R. Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):263, 2010.
5. M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. pages 289–300, 1993.
6. N. D. Jones and R. R. Hansen. The semantics of "semantic patches" in coccinelle. In *Asian Symposium on Programming Languages and Systems*, pages 303–318, 2007.
7. M. Jurczyk and G. Coldwind. Bochspwn: Identifying 0-days via system-wide memory access pattern analysis. `https://media.blackhat.com/us-13/us-13-Jurczyk-Bochspwn-Identifying-0-days.pdf`.
8. M. Jurczyk and G. Coldwind. Identifying and exploiting windows kernel race conditions via memory access patterns. Technical report, Google Research, 2013. `http://research.google.com/pubs/archive/42189.pdf`.
9. T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with intel transactional synchronization extensions. In *IEEE International Symposium on High PERFORMANCE Computer Architecture*, pages 476–487, 2014.
10. J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in OpenSSL using Coccinelle. In *European Dependable Computing Conference (EDCC)*, 2010.
11. M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. 2017.
12. H. Stuart. Hunting bugs with coccinelle. *Masters Thesis*, 2008.
13. M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1), Feb. 2005.
14. P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Usenix Security Symposium*, 2017.
15. P. Wang, K. Lu, G. Li, and X. Zhou. A survey of the doublefetch vulnerabilities. *Concurrency Computation Practice Experience*, (8):e4345, 2017.
16. F. Wilhelm. Tracing privileged memory accesses to discover software vulnerabilities. Master's thesis, Karlsruher Institut für Technologie, 2015.

17. M. Xu, C. Qian, K. Lu, B. Michael, and K. Taesoo. Precise and scalable detection of double-fetch bugs in os kernels. `http://www-users.cs.umn.edu/~kjlu/papers/deadline.pdf`.
18. R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *High PERFORMANCE Computing, Networking, Storage and Analysis*, pages 1–11, 2014.