DFTinker: Automated Detecting and Patching 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. Double-fetch bugs have attracted great attention in recent years. Double-fetch is a situation where operating system kernels fetch data from the same user address twice, and fatal errors, such as kernel crash and privilege escalation, may occur if data changes between two fetches. Prior works have studied double-fetch bugs in the Windows kernel and the Linux kernel. Bochspwn uses a dynamic method based on memory access pattern analysis, which found a series of double-fetch vulnerabilities in the Windows kernel. However, it owns a quite low code coverage and it's really time-consuming. For the Linux kernel, a static analysis method based on pattern-matching was used and succeeded to find six bugs in Linux, FreeBSD, and Android kernels. However, this method needs to check whether a double-fetch situation is a double-fetch bug manually and fix bug case by case, which cost a lot of manual efforts.

This paper proposes a hybrid approach to automatically detect and patch double-fetch bugs, based on which a prototype named DFTinker is implemented. It uses a static pattern-matching method based on Coccinelle engine to identify double-fetch bugs, achieving a more accurate result and lower false alarm rate. It also uses Intel TSX technology to avoid double-fetch bugs, and it can patch source files automatically with Coccinelle engine. According to the experiment, it owns decent performance and gets rid of lots of manual efforts. Compared with prior works, DFTinker can automatically detect and patch double-fetch bugs at the same time, and owns a high code coverage and accuracy. Furthermore, its prevention is efficacious and with a performance overhead of only 1.3%.

1 Introduction

Nowadays, various operating systems have been widely used in real-world scenarios, such as Windows, Linux, and Android. However, the careless code in operating system kernels may be utilized viciously. Double-fetch is one such careless code, which may cause information disclosure, denial-of-service, etc. [1, 6, 19]

For the sake of security, modern operating system models always isolate the kernel space and the user space to ensure system kernels won't be modified by users directly. Communication between the kernel space and the user space is achieved according to transfer functions, such as `get_user()` or `put_user()`. Double-fetch is a situation where operating system kernels read data from the same user address twice in a short time [18,19], once the data is changed between the two fetches, unexpected results even fatal errors may occur [1].

Prior works have already studied double-fetch in Windows [6, 11, 12] and Linux [15,18,19,21] systems. Bochspwn [6,11] used a dynamic approach to detect double-fetch bugs on Windows. Bochspwn defines a short time frame. Once an access to a user address happened twice in the time frame, it will be sorted as a double-fetch bug. Obviously, Bochspwn owns a low code coverage for code that under strict conditions may be never tested. Furthermore, code that Bochspwn can test is limited to its emulation ability, thus double-fetch bugs in hardware devices that it cannot emulate may be missed.

Wang *et al.* [18, 19] firstly studied double-fetch bugs in Linux kernels. He came up with a pattern-based static approach, using transfer functions to detect double-fetch bugs. Implementing with the Coccinelle engine, he successfully identified six real double-fetch bugs in Linux kernels. Besides, he proposed five solutions to deal with double-fetch bugs and achieved a patching tool using one of these solutions. The weakness of this method is that it cannot deal with double-fetch bugs introduced by compilers, such as CVE-2015-8550. And it needs lots of manual efforts to check whether a double-fetch situation is a double-fetch bug.

Xu *et al.* [21] studied double-fetch bugs in Linux kernels as well. They proposed a definition of double-fetch bugs and achieved a static analysis system named DEADLINE. Their approach needed to compile the source code to LLVM IR, and checking authenticity with a modified symbolic execution method. However, when the source code was compiled to LLVM IR, it needed to specify the target architecture, thus the system would detect only one architecture at once, leading to the miss of CVE-2016-6130. And it had common failings of the symbolic execution, such as the path explosion and constraint solving difficulties.

Schwarz *et al.* [15] proposed a method using modern CPU features to detect, exploit and eliminate double-fetch bugs. They used cache-attacks and kernel-fuzzing techniques to detect and exploit double-fetch bugs in Linux syscalls. These techniques were efficacious according to the experiment results that the exploitation success rate reached up to 97%. Schwarz *et al.* also used hardware transactional memory to eliminate double-fetch bugs. They achieved a mechanism named DropIt with a performance overhead of 0.8%. However, as their approach only applies to Linux syscalls, it will miss double-fetch bugs in other functions, such as functions in drivers. And it owns a low code coverage for its inherent characteristics.

Transactional memory [5,7,8], as an emerging parallel programming paradigm, provides opportunities to facilitate the dynamic schedules via speculative execution. Most of the time, transactional memory is used as a substitute for the lock mechanism in parallel programming [8], but recently, it has been applied to other fields as well. Guan *et al.* [3] used hardware transactional memory to protect private keys in memory, it could deal with information disclosure attacks efficaciously. Jang *et al.* [9] used hardware transactional memory to break kernel address space layout randomization, which is the core mechanism of preventing systems from memory attacks, such as buffer overflow. Transactional memory can be applied to double-fetch protection as well.

This paper proposes a hybrid approach to automatically detect and patch double-fetch bugs. Our approach consists of two phases. In the first phase, a static pattern-matching method based on Coccinelle engine is used to identify double-fetch bugs, thus it can cover all architectures in one process and achieve a more accurate result and lower false alarm rate. In the second phase, the Coccienlle engine and Intel Transactional Synchronization Extension (TSX) [22] are used to automatically patch the bug, which owns decent performance and gets rid of lots of manual efforts.

In summary, the main contribution of this paper is as follows:

- This paper proposes a hybrid approach to automatically detect and patch double-fetch bugs. The approach uses a static pattern-matching method to identify double-fetch bugs and uses Intel Transactional Synchronization Extension (TSX) to patch the bug.
- The approach was evaluated with experiments. Results show that the approach achieves a higher accuracy and lower false alarm rate for detecting double-fetch bugs. As for preventing double-fetch bugs, our approach is efficacious for preventing double-fetch bugs and its performance overhead is only 1.3.
- A prototype named DFTinker was implemented to detect double-fetch bugs and patch code automatically based on our approach. And it is now publicly available, hoping it can be useful for future study.

The rest of paper is organized as follows. Section 2 introduces background about double-fetch bugs, the Coccinelle engine, Transactional Memory, and Intel TSX. Section 3 presents the details of DFTinker, i.e., how to improve the pattern that Wang *et al.* used and how to patch code with Intel TSX technology. Section 4 shows the implementation of DFTinker and the evaluation environment. Section 5 presents the evaluation of DFTinker, including comparison with prior works, and the performance of the patched operating system. Section 6 discusses related works, limitation. The conclusion is in Section 7.

2 Background

This section will introduce related backgrounds in this paper, i.e., the principle of double-fetch bugs, Coccinelle engine, Transactional Memory, and Intel TSX.

2.1 Double Fetch

In modern operating systems, kernel space is always separated from user space for safety [17]. 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()`,

```
51   ...
52  /*
53   * Start SCLP request
54   */
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;
63    if (!sclp_ctl_cmdw_supported(ctl_sccb.cmdw))
64      return -EOPNOTSUPP;
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    }
78    rc = sclp_sync_request(ctl_sccb.cmdw, sccb);
79    if (rc)
80      goto out_free;
81    if (copy_to_user(u64_to_uptr(ctl_sccb.sccb), sccb,
          sccb->length))
82      rc = -EFAULT;
83  out_free:
84    free_page((unsigned long) sccb);
85    return rc;
86  }
87  ...
```

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

`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.

Figure 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 happens 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, malicious changes to the data between two fetches may cause unexpected results.

Wang *et al.* [18] 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 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 messages were replaced by a bigger one, it's obvious that 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

According to Section 2.1, it can figure out that double-fetch bugs and transfer functions are related closely, for each fetch means an invocation of a transfer function. Thus Wang *et al.* [18] came up with a pattern-based static analysis method, using the appearance of transfer functions to identify double fetches. He used Coccinelle engine to achieve this.

Coccinelle [10, 16] 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 [14].

One of the advantages of Coccinelle engine is path-sensitive [10], 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

As for the prevention of double-fetch bugs, the key is to keep the data fetched from user space stay the same. Wang *et al.* [18] proposed five strategies, such as checking the value consistency, overwriting data with the prior value, etc. Here we propose another way to check whether the data stay the same, i.e., using transactional memory.

Traditionally, transactional memory [5, 7, 8] 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.

A transaction is a group of operations. Before these operations executed successfully, all results will be speculative inside the transaction. Once there is a conflict during execution, the transaction will abort and revert to its initial state. It will run again until no conflict exists [7].

Lock is a traditional mechanism for parallel programming. Locks, according to the granularity of the critical sections, can be categorized as coarse-grained and fine-grained. Coarse-grained locks are easy to use but own a worse parallelism. Fine-grained locks need more efforts in programming but provide better parallelism.

Transactional memory can be divided into Software Transactional Memory (STM)and Hardware Transactional Memory (HTM). Here hardware transactional memory is chosen to implement for it owns better performance compared to software transactional memory.

Hardware transactional memory achieves transactions by processors, caches, and bus protocol [8]. It provides opportunities to implement dynamic schedules according to specific CPU instructions. Note that hardware transactional memory elides the lock and speculatively perform operations to avoid potential conflicting concurrent updates. Thus, hardware transactional memory can provide decent performance and programmer-friendly usability [13].

2.4 Intel TSX

Intel Transactional Synchronization Extensions (TSX) came out in 2013 [2,4,22], 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.

Compared with HLE, RTM is more flexible and scalable. So RTM is used to achieve prevention of double-fetch bugs.

3 Design

This section presents details about how to improve the pattern that Wang *et al.* used and how to patch code automatically with Intel TSX technology.

3.1 Detection of Double-Fetch Bugs

According to section 2.1, it 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 Figure 2 shows, they are as follows:

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

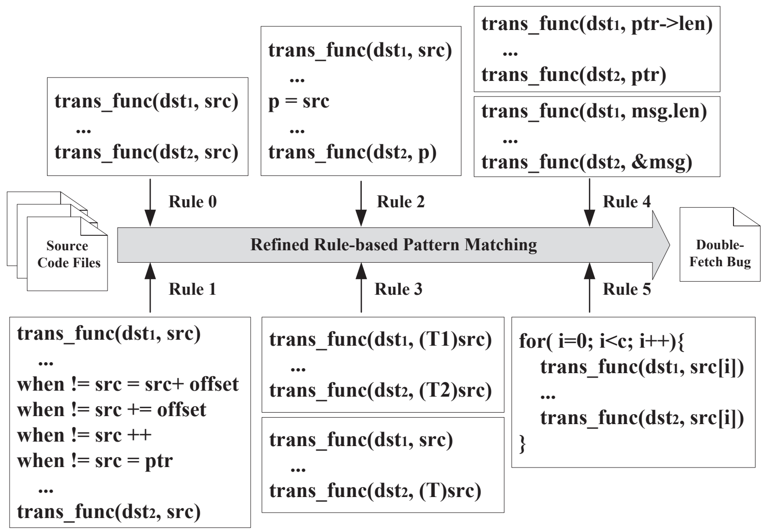

Fig. 2. Refined Coccinelle-Based Double-Fetch Bugs Detection [18]

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 Figure 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: **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

```
...
/*
 * Copy parameter control struct including a possible
 * path allocated at the end of the struct.
 */
static struct autofs_dev_ioctl *
copy_dev_ioctl(struct autofs_dev_ioctl __user *in)
{
 struct autofs_dev_ioctl tmp, *res;

 if (copy_from_user(&tmp, in, AUTOFS_DEV_IOCTL_SIZE))
   return ERR_PTR(-EFAULT);

 if (tmp.size < AUTOFS_DEV_IOCTL_SIZE)
   return ERR_PTR(-EINVAL);

 if (tmp.size > AUTOFS_DEV_IOCTL_SIZE + PATH_MAX)
   return ERR_PTR(-ENAMETOOLONG);

 res = memdup_user(in, tmp.size);
 if (!IS_ERR(res))
   res->size = tmp.size;

 return res;
}
...
```

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

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

macros playing the same role in the kernel, like `__copy_from_user_inatomic()`. In the double-fetch 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 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

```
712 ...
713 static ssize_t
714 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,
720             nfsd_net_id);
721   struct cld_net *cn = nn->cld_net;
...
728
729   /* copy just the xid so we can try to find that */
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;
755
756   wake_up_process(cup->cu_task);
757   return mlen;
758 }
759 ...
```

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

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 Figure 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 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 Figure 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

```
int isdn_ppp_write(int min, struct file *file,
    const char __user *buf, int count)
{
  isdn_net_local *lp;
  struct ippp_struct *is;
  int proto;
...
  if (!lp)
    printk(KERN_DEBUG "isdn_ppp_write:␣lp␣==␣NULL\n");
  else {
    if (lp->isdn_device < 0 || lp->isdn_channel < 0) {
...
      if (copy_from_user(protobuf, buf, 4))
        return -EFAULT;

      proto = PPP_PROTOCOL(protobuf);
      if (proto != PPP_LCP)
        lp->huptimer = 0;

      return 0;
    }

    if ((dev->drv[lp->isdn_device]->flags &
    DRV_FLAG_RUNNING) &&
...
      skb_reserve(skb, hl);
      cpy_buf = skb_put(skb, count);
      if (copy_from_user(cpy_buf, buf, count))
      {
        kfree_skb(skb);
        return -EFAULT;
      }
...
  }
  return count;
}
```

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

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 file `\drivers\isdn\hysdn\hysdn_procconf.c` is matched with double-fetch 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 situation.

3.2 Automated Patching with Intel TSX

As for prevention, Wang *et al.* proposed five methods to avoid double-fetch bugs. They are : **(1) Don't Copy the Header Twice.** Double-fetch bugs can be thoroughly avoided by changing into one fetch. If there is only one fetch, any malicious change to data will be useless for they won't be fetched into kernel

```
shared variable:
    mutex

local variable:
    retries

procedure LOCK( )
    retries ← 0
    _xbegin( )                                        ▷ speculative path
    if lock_is_free(lock) then
        return
    else
        _xabort( )
    end if
    retries++
    if retries < MAX_RETRIES then
        goto line 9
    else
        mutex.lock( )                                 ▷ fallback path
    end if
end procedure

procedure UNLOCK( )
    if _xtest( ) then
        _xend( )
    else
        mutex.unlock( )
    end if
end procedure
```

Fig. 6. Speculative lock & unlock with HTM

anymore; **(2) Use the Same Value.** A double-fetch bug can be harmful when the data changed after its validity been checked. This can be solved by using the same value fetched by the first, i.e., ignore the data got by the second fetch; **(3) Overwrite Data.** Overwriting data which may be changed by the malicious user is also a useful solution. This is always used when the first fetch is a message header, and it overwrites message header after fetching the whole message. This method is widely used in FreeBSD code; **(4) Compare Data.** Adding a compare operation after the second fetch is another solution. Once the data fetched twice are different, prevention measures will work; **(5) Synchronize Fetches.** The last method is using synchronized fetches. Traditional synchronization mechanisms used in parallel programs, such as locks, are suitable for resolving double-fetch bugs. They will provide a guarantee of data consistency in the user space. However, this method will sacrifice performance of the system, making itself worthless.

Table 2. Results of Detection of Double-Fetch bugs

Kernel	Version	Files	Cases
Linux	4.14.10	45614	24
FreeBSD	11.1	38811	13
OpenBSD	6.2	29704	0
Android	8.1.0 (3.18)	30479	0

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 patching 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 shown in Figure 6, 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 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 Figure 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.

As for the source code, the experiment used Linux 4.14.10, OpenBSD 6.2, FreeBSD 11.1, and Android 8.1.0. These were the newest version when the experiment was conducted. To prove that DFTinker can find out bugs reported 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 patched operating system, another Ubuntu 14.04 x64 was implemented, whose function `ioctl_file_dedupe_range()` in file `/fs/ioctl.c` was patched by 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 patched by DFTinker.

5.1 Detection of Double-Fetch Bugs

At first, DFTinker was run to confirm Wang *et al.*'s work [18] 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 four open source kernels: Linux 4.14.10, FreeBSD 11.1, OpenBSD 6.2, Android 8.1.0 (kernel version 3.18). These were the newest version when the experiment was conducted. The result is shown as Table 2.

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 patched, but it still found 24 cases in total. The details are shown in Table 3.

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, 13 cases were found in total, they were shown in Table 3.

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 8.1.0 based on Linux

Table 3. Results of Detection of Linux 4.14.10 and FreeBSD 11.1

No.	File	Function	First Fetch	Second Fetch
1	bus.c	__nd_ioctl()	942	1025
2	commctrl.c	ioctl_send_fib()	82	119
3	compat.c	cmsghdr_from_user_compat_to_kern()	138	167
4	core.c	sched_copy_attr()	4342	4381
5	core.c	perf_copy_attr()	9639	9676
6	custom_method.c	cm_write()	37	54
7	dev-ioctl.c	copy_dev_ioctl()	100	109
8	dir.c	ll_dir_ioctl()	1485	1504
9	dpt_i2o.c	adpt_i2o_passthru()	1734	1834
10	hpioctl.c	asihpi_hpi_ioctl()	131	140
11	ioctl.c	ioctl_file_dedupe_range()	586	597
12	llite_lib.c	ll_copy_user_md()	2463	2478
13	megaraid_mm.c	mega_m_to_n()	3443	3467
14	mpt3sas_ctl.c	_ctl_ioctl_main()	2261	2311
15	nfs4recover.c	cld_pipe_downcall()	730	753
16	opal-prd.c	opal_prd_write()	238	244
17	psdev.c	coda_psdev_write()	109	128
18	scsi_ioctl.c	sg_scsi_ioctl()	440	466
19	tls_main.c	do_tls_setsockopt_tx()	351	379
20	uhid.c	uhid_event_from_user()	407	455
21	util.c	strndup_user()	187	195
22	vhost.c	vhost_vring_ioctl()	1361	1379
23	vt.c	con_font_set()	4135	4152
24	wext-core.c	ioctl_standard_iw_point()	747	809
25	aac.c	aac_ioctl_sendfib()	2999	3007
26	aacraid.c	aac_ioctl_sendfib()	2763	2771
27	bcm2835_vcio.c	vcio_ioctl()	66	72
28	dtrace.c	dtrace_dof_copyin()	13210	13234
29	fasttrap.c	fasttrap_ioctl()	2277	2294
30	freebsd32_misc.c	freebsd32_jail()	2300	2311
31	hwpmc_x86.c	pmc_save_user_callchain()	112	128
32	kern_jail.c	sys_jail()	263	274
33	linux_futex.c	linux_sys_futex()	816	819
34	netmap_pt.c	ptnetmap_read_cfg()	779	790
35	oce_if.c	oce_handle_passthrough()	2293	2305
36	sys_capability.c	sys_cap_rights_limit()	259	267
37	usb_generic.c	ugen_fs_copy_in()	1108	1202

kernel version 3.18. Unfortunately, it didn't found double-fetch bugs on Android as well.

5.2 Automated Patching with Intel TSX

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

To evaluate the overhead performance of the patched code, an Ubuntu 14.04 x64 was used, whose kernel version was 4.6.1. DFTinker was used to patch the function `ioctl_file_dedupe_range()` in the file `/fs/ioctl.c`, which was reported as CVE-2016-6516. 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 patched. After testing for twenty times and taking the average, the patched operating system owned a decent overhead performance of 1.3 %.

6 Discussion

This section will discuss related works about double-fetch bugs and limitation of DFTinker.

6.1 Related Work

Both double-fetch bugs and transactional memory are widely studied nowadays.

Double-fetch bugs. Similar to program analysis, methods used to study double-fetch bugs can be divided into dynamic and static.

Jurczyk and Coldwind [11,12] used a dynamic approach to study double-fetch bugs. By tracing memory accesses, they successfully found double-fetch bugs in the Windows kernel. Inherently, this approach owned a low code coverage, i.e., code under strict conditions may never be tested. Furthermore, code that this method can test is limited to its emulation ability, thus double-fetch bugs in hardware devices that it cannot emulate may be missed.

Wang *et al.* [18] was the first to study double-fetch bugs in Linux kernel systematically. He came up with a pattern-based static approach, using transfer functions to detect double-fetch bugs. Implementing with the Coccinelle engine, he successfully identified six real double-fetch bugs in Linux kernels. Besides, he proposed five solutions to deal with double-fetch bugs and achieved a patching tool using one of these solutions. The weakness of this method is that it needs lots of manual efforts to check whether a double-fetch situation is a double-fetch bug. And because of the incomplete model of the double-fetch bug, its accuracy is undesirable and the false alarm rate is quite high.

Xu *et al.* [21] studied double-fetch bugs in Linux kernels as well. They proposed a definition of double-fetch bugs and achieved a static analysis system named DEADLINE. Their approach needed to compile the source code to LLVM IR, and checking authenticity with a modified symbolic execution method. However, when the source code was compiled to LLVM IR, it needed to specify the target architecture, thus the system would detect only one architecture at once, leading to the miss of CVE-2016-6130. And it had common failings of the symbolic execution, such as the path explosion and constraint solving difficulties.

Schwarz *et al.* [15] proposed a method using modern CPU features to detect, exploit and eliminate double-fetch bugs. They used cache-attacks and kernel-fuzzing techniques to detect and exploit double-fetch bugs in Linux syscalls. These techniques were efficacious according to the experiment results that the exploitation success rate reached up to 97%. Schwarz *et al.* also used hardware transactional memory to eliminate double-fetch bugs. They achieved a mechanism named DropIt with a performance overhead of 0.8%. However, as their approach only applies to Linux syscalls, it will miss double-fetch bugs in other functions, such as functions in drivers. And it owns a low code coverage for its inherent characteristics.

In our approach, we used a source-level static analysis method, which could cover all architectures in one process and owned a high code coverage. Besides, our better model led to a higher accuracy and lower false alarm rate.

Transactional memory. Transactional memory [5,7,8], as an emerging programming paradigm, has attracted great attentions. With transactional memory, programmer can easily implement fine-grained operations.

Most of the time, transactional memory is used as a substitute for the lock mechanism in parallel programming [8], but recently, it has been applied to other fields as well.

Guan *et al.* [3] used hardware transactional memory to protect private keys in memory, it could deal with information disclosure attacks efficaciously. Jang *et al.* [9] used hardware transactional memory to break kernel address space layout randomization, which is the core mechanism of the preventing systems from memory attacks, such as buffer overflow.

The transactional memory is successfully applied to double-fetch protection as well. With its feature, DFTinker can guarantee the consistency of data between two fetches, which can solve double-fetch bugs radically.

6.2 Limitation

Although DFTinker achieves a decent performance in detecting and patching double-fetch bugs, it still owns a few weaknesses.

- DFTinker cannot find out double-fetch bugs with unknown transfer functions. In fact, there are many normal functions contains one or more transfer functions, so these normal functions can be regarded as transfer functions, too. However, it is not easy for a static method to identify whether a function call transfer functions finally or not. This is left for future work.
- DFTinker cannot figure out double-fetch bugs introduced by compilers, like CVE-2015-8550 [20]. As DFTinker works on source code level, any double-fetch bug happens in lower level cannot be found. It may be detected by a dynamic approach. Nevertheless, DFTinker can prevent such bugs with our prevention approach based on transactional memory.

7 Conclusion

In this paper, we implemented a prototype named DFTinker, which could detect double-fetch bugs and patch code automatically. With more transfer functions, DFTinker identified more double-fetch situations and owned a higher accuracy. With stricter rules, DFTinker filtered double-fetch bugs where it couldn't result in fatal errors, lowering false alarm rate significantly. In total, DFTinker detected 24 cases in Linux 4.14.10, 13 cases in FreeBSD 11.1, and no case in OpenBSD 6.2 and Android 8.1.0. Finally, DFTinker used hardware transactional memory to prevent double-fetch bugs, which is efficacious and with a performance overhead of only 1.3%.

References

1. Bug 166248 – CAN-2005-2490 sendmsg compat stack overflow.
2. 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.
3. L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *IEEE Symposium on Security and Privacy*, pages 3–19, 2015.
4. F. Haas and S. Metzlaff. Enhancing real-time behaviour of parallel applications using intel tsx. *Entomological Science*, 11(1):123126, 2014.
5. 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.
6. S. Hammou. Exploiting Windows drivers: Double-fetch race condition vulnerability, 2016.
7. T. Harris, J. Larus, and R. Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):263, 2010.
8. M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. pages 289–300, 1993.
9. Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. In *ACM Sigsac Conference on Computer and Communications Security*, pages 380–392, 2016.
10. 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.
11. 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`.
12. 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`.
13. 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.

14. 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.
15. 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.
16. H. Stuart. Hunting bugs with coccinelle. *Masters Thesis*, 2008.
17. 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.
18. 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.
19. P. Wang, K. Lu, G. Li, and X. Zhou. A survey of the doublefetch vulnerabilities. *Concurrency Computation Practice Experience*, (8):e4345, 2017.
20. F. Wilhelm. Tracing privileged memory accesses to discover software vulnerabilities. Master's thesis, Karlsruher Institut für Technologie, 2015.
21. 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`.
22. 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.