Front.Comput.Sci.xxx
DOI10.1007/s11704 − 016 − 6383 − 8

RESEARCH ARTICLE

DFTracker: Detecting Double-fetch Bugs by Multi-taint Parallel Tracking

Pengfei WANG (✉)[1,2,3], Kai LU [1,2,3], Gen LI [1,2,3], Xu ZHOU [1,2,3]

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

Abstract A race condition is a common trigger for concurrency bugs. As a special case, a race condition can also occur across the kernel and user space causing a double-fetch bug, which is a field that has received little research attention. In our work, we first analyzed real-world double-fetch bug cases and extracted two specific patterns for double-fetch bugs. Based on these patterns, we proposed an approach of multi-taint parallel tracking to detect double-fetch bugs. We also implemented a prototype called DFTracker (double-fetch bug tracker), and we evaluated it with our test suite. Our experiments demonstrated that it could effectively find all the double-fetch bugs in the test suite including eight real-world cases with no false negatives and minor false positives. In addition, we tested it on Linux kernel and found a new double-fetch bug. The execution overhead is approximately 2X for single-file cases and approximately 9X for the whole kernel test, which is acceptable. To the best of the authors' knowledge, this work is the first to introduce multi-taint parallel tracking to double-fetch bug detection—an innovative method that is specific to double-fetch bug features—and has better path coverage as well as lower runtime overhead than the widely used dynamic approaches.

Keywords Multi-taint Parallel Tracking; Double Fetch; Race Condition between Kernel and User; Time of Check to Time of Use; Real-world Case Analysis; Clang Static Analyzer

Received Jul. 22, 2016; accepted Dec. 8th, 2016

E-mail: pfwang@nudt.edu.cn

1 Introduction

In recent years, with the wide application of multi-core technology in hardware, multi-thread programs have been increasingly used, and the reliability of concurrent programs has become a significant issue. As concurrent processing is primarily designed to improve process speed, concurrent programming is pervasively used in large-scale multi-functional software that has intensive computation and real-time demand, such as industrial control software, operating systems, and complex applications. Therefore, concurrent program reliability affects not only daily life applications but also social stability and national security.

A race condition is a situation where a shared object is accessed concurrently but without proper synchronization to force the ordering. Historically, software errors caused by race conditions led to many significant security incidents, such as the Therac-25 accident [1], the 2003 North American blackout [2], and the 2012 Facebook IPO delay [3], all of which resulted in significant casualties or loss of property, having profound impacts. A race condition is a typical trigger for concurrency bugs, which varies significantly and can occur with different granularity throughout the entire system. For example, racing between common threads, i.e., data race [4] [5] [6]; racing between system processes [3]; racing between hardware devices [1];

racing between different library methods invocation [7]; racing when accessing the file system [8]; racing between events in event-driven mobile applications [9] [10]; and even beyond a single operating system, such as racing between a hypervisor and guest virtual operating system [11]; racing between different computing nodes in a distributed computing cluster [12]. The most general case is a data race, which is the race condition among user threads. A data race occurs when two threads are about to access the same memory location, and 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 [13].

In addition to the above cases, a race condition can occur across the kernel and user space. This situation can cause a special case of time-of-check to time-of-use (TOCTOU) issue, and Serna [14] introduced the term "double fetch" to describe it. A double-fetch bug occurs when the kernel (e.g., via a syscall) reads the same value that resides in the user space twice, first time for verifying or establishing a relation with the kernel and the second time for use. Meanwhile, a concurrently running user thread changes the value under a race condition. If the value is modified within the time window between the two kernel reads, a data inconsistency occurs, which may lead to grave consequences for the kernel, such as buffer overflows, privilege escalation, information leakage, or even kernel crash. Jurczyk and Coldwind [15] were the first to study the double-fetch issue systematically in their Bochspwn project. They detected double-fetch issues based on memory access tracing and discovered a series of double-fetch bugs in the Windows kernel. However, their dynamic approach is limited in the coverage it can achieve, and we cannot apply it to the code that needs specific hardware to be executed. For example, it cannot analyze a driver without having the driver's device.

Double-fetch situations are pervasive in the real world, including platforms such as Windows, Linux, and FreeBSD. In addition, double-fetch bugs occur not only in the source code but also in macros [16]. They could also be introduced by the compiler during compiling [17], which are difficult to detect and prevent. Several approaches have been proposed for race condition detection at the memory access level. Static approaches analyze the program without running it [18] [19] [20] [21] [22]. They can find the corner cases because of the better coverage of the code and the overall knowledge of the program. However, the major disadvantage is the false reports generated owing to the lack of the program's full runtime execution context. Dynamic approaches execute the program to verify race conditions [23] [24] [4], checking whether a race could cause program failure in executions. They control the active thread scheduler to trigger specific interleaving to increase the probability of bug manifestation [25]. Nevertheless, runtime overhead is a severe problem, and it could even impose more than 20x overhead when detecting memory-intensive programs [26] [27]. Unfortunately, none of the existing approaches can be applied to double-fetch bug detection directly, and the reasons are as follows.

- A double-fetch bug is caused by the race condition between kernel and user space, which is different from a typical data race because the race condition is separated by the kernel and user space. For a typical data race, the read and write operations exist in the same address space. Therefore, most of the previous approaches detect data races by identifying both the read and write operations that access the same memory location. However, it is different for a double-fetch bug. The kernel side only contains two reads whereas the write resides in the user thread. Moreover, the write of the user thread is potential, which means it does not essentially exist when analyzing the program, but can be created by a malicious user to trigger the race condition.
- The involvement of the kernel makes a double-fetch bug different in the way it accesses data. Owing to the isolation of virtual memory spaces, the kernel fetching data from the user space to kernel space relies on a specific internal scheme (e.g., copy functions in Linux: `copy_from_user()`, `get_user()`), rather than dereferencing the user pointer directly, which means the common data race detection approaches based on pointer dereference are no longer applicable.
- Moreover, a double-fetch bug is rather complicated as a semantic bug. A double-fetch bug requires a fetch (the check) phase and a use phase (where the fetched data is used). Although the check can be located by matching the patterns of fetch operations, the use of the fetched data varies considerably. Therefore, we need to consider the double-fetch characteristics, and previous approaches cannot be adopted directly.
- Finally, as a special case of TOCTOU bug, a double-fetch bug can turn into a double-fetch vulnerability once the caused result is exploitable, such as buffer overflow, information leakage, and kernel crash. A typical data race is caused by specific thread interleaving with inappropriate synchronization,

whereas a double-fetch bug might be facing the race condition crafted by a malicious user, who has a clear purpose and could cause more severe results.

Due to the above reasons, the double-fetch issue is special and previous approaches are neither applicable nor effective. As the double-fetch issue is a relatively new research area, few systematic works have been performed on double-fetch bug detection except Bochspwn. Thus, in this paper, we propose an innovative approach named multi-taint parallel tracking that is specific to double-fetch bug detection. Our approach is based on the extracted double-fetch bug patterns and specific to double-fetch features. It requires no hardware, which is suitable for code analysis such as drivers. In addition, our static-based approach is path-sensitive, which achieves better coverage than previous dynamic approaches and with lower overhead. Overall, our main contributions include the following.

- **Extraction of double-fetch bug specific patterns**. As per the analysis of real-world double-fetch bugs, we extracted two double-fetch bug patterns based on program execution path and branch condition, which are specific to the feature as well as semantic of double-fetch bugs.
- **Multi-taint parallel tracking approach**. Based on the extracted double-fetch bug patterns, we proposed an approach named multi-taint parallel tracking to detect double-fetch bugs. To the best of the authors' knowledge, this work is the first to introduce multiple-taint parallel tracking to double-fetch bug detection, which maps the time factor of memory access ordering to the space factor of different tainted variables, specific to double-fetch bug features.
- **Implementation and evaluation of the proposed approach**. We implemented a prototype of our proposed approach called DFTracker (double-fetch bug tracker), which is a checker of the Clang Static Analyzer. Experiments have proved its feasibility and viability. DFTracker could successfully find all the double-fetch bugs from the test cases as well as some existing real-world double-fetch bugs, with no false negatives and minor false positives. The overhead is no more than 2X.
- **Discovery of a new double-fetch bug**. We tested DFTracker on Linux kernel-3.18, and found a new double-fetch bug in file `/fs/fhandle.c`. The execution overhead is approximately 9X for the full-path exploration of the entire kernel, which is acceptable. We provide a detailed analysis of this new bug in Section 6.1.

The rest of the paper is organized as follows. In Section 2, we present the background and introduction to issues that are related to double fetch. In Section 3, we analyze some real-world double-fetch bug cases, extract two patterns, and propose our new approach based on these basic patterns. In Section 4, we describe the design and implementation of the prototype. In Section 5, we evaluate our work with some test experiments and compare the results with the work of others. In Section 6, we present a detailed analysis of the new bug and provide some useful advice on double-fetch bug prevention. Section 7 surveys the related work. Finally, we present the conclusions of our work in Section 8.

2 Background

2.1 Copy Scheme between Kernel and User Space

In modern computer systems, the memory is divided into the kernel space and the user space. The kernel space is where the kernel code is stored and executes, whereas the user space is where regular user processes run. The kernel space and the user space are independent and implemented in separate address spaces. As the virtual address spaces are mapped to a considerably smaller physical address space through page tables, pages are dynamically swapped in and out to improve the utilization of the physical memory. This virtual memory model isolates each address space, including the kernel space. In the user space, a shared memory region can be created and shared between two or more processes to facilitate the inter-process communication. Each process maps the shared memory region to a different address in their respective address spaces, and data from the other address space can be accessed by directly dereferencing a pointer. However, things are different between the kernel space and the user space. As the kernel address space and the user address space are both virtual and physically isolated, shared memory access is not feasible. Therefore, different copy schemes are provided by the kernel to exchange data between the kernel space and user space. For example, some specific copy functions are created to perform this job in Linux, such as `copy_from_user()`, `copy_to_user()`, or macros such as `get_user()` and `put_user()`.

Copy functions not only exchange data between the kernel and the user space but also provide protection

mechanisms on memory accessing. The return value indicates whether the copy operation is successful, and exceptions such as illegal address access or page faults are handled properly. Copying data through these copy functions is the only way to communicate between the kernel space and user space in Linux. Therefore, any occurrence of double-fetch issue should involve invocations of these copy functions.

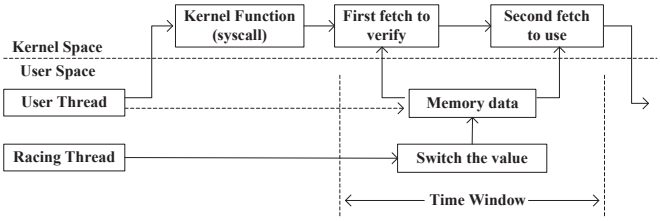


Fig. 1 Illustration of how a double-fetch bug occurs

2.2 TOCTOU

A TOCTOU issue (or bug) is caused by changes between the checking of a condition and the use of the result of that check (by which the condition no longer holds). When a program checks for a particular characteristic of an object (or event) to take action based on the assumption that the characteristic still holds, if some action during that interval invalidates the assumption, the results of the second action may not be what was intended [28]. The invalidation of the assumption (data inconsistency) in TOCTOU is usually caused by a race condition, which results from improper synchronized concurrent accesses to a shared object. There are varieties of shared objects in a computer system, such as files [28], sockets [29], and memory locations [30]. Therefore, a TOCTOU issue can exist in different layers throughout the system.

TOCTOUs are common in Unix file systems, which can date back to the 1990s. During the past decades, numerous approaches [31] [32] [33] [8] [34] have been proposed to solve this problem, but there is still no general, portable, secure way for applications to access the file system in a race-free way due to the complexity of this issue. These approaches have to consider many specific aspects of this issue, such as the mapping between filenames and inode, the resolution of symbolic links, the UNIX file-system interface and implementation. Static analysis for file-system races results in numerous false positives and negatives, whereas kernel-based runtime race detectors and preventers are operating system specific and therefore may not catch all types of races.

2.3 Double Fetch

A double fetch is a special case of TOCTOU that occurs in memory access level involving the kernel. There is no actual shared object but multiple reads of the same user space data from the kernel. As illustrated in Fig. 1, a double-fetch bug occurs within a kernel function, such as a system call, which is invoked by a user application from user mode. The kernel fetches a value from user space the first time to verify or to establish the relation between kernel objects, and then fetches it again to use, whereas a concurrently running user thread changes the value under a race condition. The inconsistency caused between the two fetches may violate some assumptions made by the kernel based on the first fetch, which is likely to cause serious problems. Moreover, the involvement of the kernel worsens the consequences of a double-fetch bug. A double-fetch bug can turn into a double-fetch vulnerability once the consequence is exploitable, such as buffer overflow, information leakage, and kernel crash.

As a special case of TOCTOU issue that occurs in the memory access level, a double-fetch bug focuses on the interaction between the kernel and user space, which means previous TOCTOU detection approaches specified for the file system or other shared objects are no longer applicable. Although dynamic approaches are easy to conduct, problems such as path coverage and hardware requirement restrict the usefulness. In addition, instrumenting the kernel introduces huge runtime overhead that is sometimes unbearable, and it is also likely to influence the functionality of the kernel and affect the results.

A double-fetch bug is a special TOCTOU issue that occurs when the kernel is accessing user memory space, which is triggered by a race condition between the kernel and user space. In other words, a double-fetch bug combines a race condition with a TOCTOU bug. The race condition between the kernel and the user space plays a significant role in causing a double-fetch bug. Even though the traditional TOCTOU can be a single-thread pattern, the double-fetch issue has to be multi-threaded. Therefore, a double-fetch bug is different from previously known cases that are specific to the shared objects (e.g., a file or a socket). We propose an approach that is specific to the double-fetch bug feature of accessing user space memory from kernel space.

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

Fig. 2 Double-fetch bug in CVE-2008-2252

3 Pattern-Based Double-Fetch Bug Detection

3.1 Real-World Case Study

Double-fetch bugs have existed for a very long time; however, owing to the uncertainty of appearance and the lack of systematic study, only a few were discovered and reported. We collected as many cases as possible and finally obtained 11 real-world cases to carry out our study. Owing to restrictions in article length, we only present three representative cases here to demonstrate how a double-fetch bug occurs.

CVE-2008-2252: Here, we present a double-fetch bug case in win32k.sys, which had been patched in MS08-061 [14]. In this case, an inadequate pool allocation and a later memory pool overflow can be caused, which might lead to the execution of arbitrary code in kernel mode. We can see from Fig. 2, `my_struct->cbData` is fetched twice at line 5 and line 9. However, the constraint at line 8, which aims to guarantee the successful pool allocation, is no longer effective after the second fetch of `my_struct->cbData` (line 9), causing the potential inconsistency between the check (branch at line 8) and the use of the fetched value (line 9). Therefore, a concurrently running thread may have the chance to switch `my_struct->cbData` to a large value, which will result in pool overflow.

CVE-2005-2490: In Linux kernel 2.6.9, when we copy user control content to kernel using the function `sendmsg()` in file `compat.c`, the same user data are accessed twice without sanity check at the second time [35]. This can cause a kernel buffer overflow and therefore could lead to privilege escalation. As shown in Fig. 3, this function works in two steps: examining the parameters in the first loop (checked at line 8) and copying the user data in the second loop (used at line 19). However, only the first fetch

```
1  int cmsghdr_from_user_compat_to_kern(struct msghdr *kmsg, unsigned
char *stackbuf, int stackbuf_size){
2    ...
3    while(ucmsg != NULL) {
4       if(get_user(ucmlen, &ucmsg->cmsg_len))   // 1st fetch
5          return -EFAULT;
6       if(CMSG_COMPAT_ALIGN(ucmlen) <
CMSG_COMPAT_ALIGN(sizeof(struct  compat_cmsghdr)))
7          return -EINVAL;
8       if((unsigned long)(((char __user *)ucmsg – (char __user*) kmsg->
msg_control) + ucmlen) > kmsg->msg_controllen) // check
9          return -EINVAL;
10      ...
11      ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
12   }
13   ...
14   while(ucmsg != NULL) {
15      __get_user(ucmlen, &ucmsg->cmsg_len); //2nd fetch
16      tmp = ((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
CMSG_ALIGN(sizeof(struct cmsghdr)));
17      kcmsg->cmsg_len = tmp;
18      ...
19      if(copy_from_user(CMSG_DATA(kcmsg),CMSG_COMPAT_DATA
(ucmsg), ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg)))) //use
20         goto out_free_efault;
21      kcmsg = (struct cmsghdr *)((char *)kcmsg + CMSG_ALIGN(tmp));
22      ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
23   }
24   ...
```

Fig. 3 Double-fetch bug in CVE-2005-2490

(line 4) of the parameter `ucmlen` is examined (line 8), whereas the second fetch (line 15) is free from constraint, which may cause potential data inconsistency between the check and the use. This could cause buffer overflow in the copy function (line 19) if the parameter `ucmlen` is modified by another concurrently running thread at the user side.

Preprocessed macro: A double-fetch bug can also occur in the preprocessing stage such as macro expansion, and we give an example here [16] in file `list.h` of Linux3.9-rc1, which had been patched in rc-3. As shown in Fig. 4, function `hlist_entry_safe()` fetches the pointer `ptr` twice in the ternary operator (line 3), the first time to test for nullity and the second time to pass it as an argument to `hlist_entry()`, which computes the offset back to the enclosing structure based on `ptr`. Therefore, the use of `ptr` in the second fetch is based on the check of `ptr` in the first fetch; only when `ptr` passes the nullity test can it be used in

```
1 #ifndef  hlist_entry_safe
2 #define  hlist_entry_safe(ptr, type, member) \
3    (ptr)? hlist_entry(ptr, type, member) : NULL
4 #undef  hlist_for_each_entry
5 #define hlist_for_each_entry(pos, head, member)
```

Fig. 4 Double-fetch bug in a macro definition

```
  ...
1 a = read(addr,len)          // first fetch to verify
2 ...
3 if Condition(a)             // branch condition controlled by first fetch
4    ...
5    a = read(addr,len)       // second fetch to use
6    ...
7    memcpy(dst,src,a)        // use of a, from second fetch
8    ...
```

Pattern 1

```
  ...
1 a = read(addr,len)          // first fetch pass to b
2 b = f(a)
3 ...
4 a = read(addr,len)          // second fetch to verify
5 ...
6 if Condition(a) :           // branch condition controlled by second fetch
7    ...
8    memcpy(dst,src,b)        // use of b, passed by first fetch
     ...
```

Pattern 2

Fig. 5 Double-fetch bug pattern 1 and pattern 2

`hlist_entry()`. However, the pointer is likely to be changed between the two fetches under a race condition, which could lead to a NULL-pointer crash when `ptr` is used in `hlist_entry()`. In addition, this occurs in the preprocessing stage, and would be hard to discover using a regular approach.

3.2 Double-Fetch Bug Patterns

After analyzing the real-world double-fetch bug cases, we can conclude that a double-fetch bug occurs because the programmer uses a newly fetched value instead of the previously verified one. As a matter of fact, there should always be some restrictions of the function arguments, such as the pointer should not be NULL and the length should not exceed the buffer size. In general, programmers should verify the function arguments before using them. However, problems occur when they fail to use the arguments that have been verified. Instead, they are likely to retrieve the arguments from user space one more time, which causes a double-fetch bug. This is more likely to occur when the source code is complicated and long. Cases in which the user pointer instead of the fetched value is passed to other function invocations also increase the occurrence.

To characterize a double-fetch bug more precisely, we abstract some critical elements from the known double-fetch bug analysis, which are useful for the pattern-based detection approach. These elements are listed as follows.

- **A kernel function**: where a double fetch takes place.
- **A first kernel read**: reads a value from the user space. The value fetched by this read is used to verify or establish some relation.
- **A branch condition**: also known as the "*check*," which is controlled by the first fetched value. Only when this condition is satisfied will the second fetch and the use of the fetched value take place.
- **A second kernel read**: fetches the same value as the first read or part of the data struct overlaps. The newly fetched value is used later without another sanity check.
- **Use of the fetched value**: also known as the "*use*," which occurs after the second read and invalid value may cause serious consequences.

As we can see from the elements listed, the branch condition is a linkage that bridges the two kernel reads and steers the execution toward the path that errors can be triggered if the user data are modified. The second read is the source of the bug, which breaks the consistency. The *use* is the trigger of the bug, and no problems will occur if the second fetched value is never used. We can conclude that a potential double-fetch bug occurs in the following way.

The first fetched value controls a branch condition, which is related to the verification of the value. Within this branch, which means when the verification is passed, a second fetch of the same value occurs, followed by the use of the newly fetched value but without an additional check. This situation covers most occurrences of the real double-fetch bugs.

Based on the double-fetch elements we extracted and the above description, we propose two double-fetch bug patterns, which are shown in Fig. 5. In pattern 1, a kernel read first fetches a value and assigns it to variable `a` for verification (line 1). If the value is valid, then it satisfies the branch condition (line 3). Within this branch, a second kernel read takes place (line 5), and the newly fetched value is used (line 7), which may be a serious result owing to data inconsistency. Pattern 2 is similar to pattern 1, except for the ordering difference. A kernel read first fetches a value (line 1), but this value is not used to verify but passed to another variable `b` (line 2). Then, a second kernel read occurs, which fetches the value again, and assigns it to variable `a` to verify (line 4). If it satisfies the branch condition (line 6), then it comes to the use phase (line 8). However, in this pattern, it is the old value `b` fetched the first time that is used by the kernel, which may cause serious errors owing to data inconsistency.

In summary, we propose two double-fetch bug patterns, which are based on the extracted elements. In pattern 1, a

valid value is changed to an invalid one after passing the sanity check but before using it. In pattern 2, an invalid value is changed to a valid one in the second fetch to pass the sanity check; however, it is still the invalid one that is used. Both situations can cause serious double-fetch bugs, and these two patterns are used by the pattern-based detection approach we propose.

3.3 Multi-Taint Parallel Tracking

A double-fetch bug occurs because the inconsistency of the user data is violated between the two reads. Thus, the basic idea of our approach is assuming that all the user data fetched by kernel reads are UNTRUSTED , namely any newly fetched data are inconsistent from the last fetch. In other words, any second fetch of the same value obtains a different one, which has already been changed by a potentially malicious user thread.

With the help of taint tracking and symbolic execution, we can analyze how the untrusted data propagates from being introduced to being used. In our approach, a *Taint* is defined as follows:

Taint = <*number*, *time*, *origin*>

where: *number* is the number tag of a taint, which is generated and added to a variable when the kernel is fetching new data from the user space; *time* indicates the timestamp when this taint is introduced (by creation or propagation); *origin* denotes the original memory address that the user data come from, which is a user space address where the kernel read takes place. In the implementation level, a taint is a structural tag that is added to the symbolic expression of a variable.

Therefore, we taint all the untrusted data read by the kernel from the user space. Moreover, to distinguish where the data come from, how the data propagate in the program, and how they affect the execution, we add different taints (different tag numbers) to data fetched by different kernel reads. All the taints propagate independently, and all the values generated by the following arithmetic computation and logical computation during the tracking also inherit the tainted property. As shown in Fig. 6 , we use a new taint every time a kernel read takes place and propagate all the taints concurrently while we perform the symbolic execution. Our approach maps the time factor of memory access ordering to the space factor of different tainted variables, which is specific to double-fetch characteristics.

Based on the patterns we described in the last subsection, we detect double-fetch bugs by checking taints and

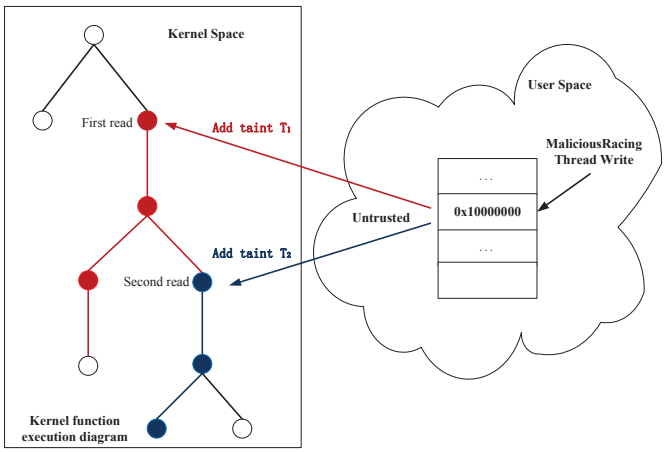


Fig. 6 Multi-taint parallel tracking overview

branches. Each time a tainted value is used, we check whether this occurs within a branch controlled by different taints from the same user space address. If yes, then the check and the use of that value are obtaining data from different read operations. Thus, a potential double-fetch bug is indicated.

We now illustrate our method by a typical double-fetch bug, which is especially common among device drivers. As shown in Fig. 7 , in most situations, the kernel (e.g., syscalls or drivers) needs to receive the message (a predesigned data struct) from the user space, and send back the message after processing it. However, the length of the message is often unfixed, which means the kernel has to check the message length first by only copying a fixed-length header to the kernel (line 3) to access the `size` field of that header. Then, the kernel allocates a buffer based on the length of the whole message (line 7) and copies in the entire message by a second copy (line 11). Finally, after being processed, the message is copied back to the user space (line 15). A double-fetch situation is inevitable when copying an unfixed-length message between the kernel and the user space. This double-fetch situation turns into a double-fetch bug once the `size` field of the header in the second copied message is used again (line 15), because `size` might be changed between the two copies and consequences relevant to `size` might have occurred, such as kernel information leakage, buffer overflow, or over-boundary memory access.

We apply our multi-taint parallel tracking method to this example. When the first fetch takes place, we add taint `T1` to the fetched value `hdr` (line 3). Therefore, `size` field of `hdr` is tainted by `T1`, and branch condition at line 5 is also controlled by `T1`. Then, taint `T1` propagates to `msg` by the

```
1 void kernel_func( *uptr){
2   ...
3   copy_from_user(hdr, uptr, sizeof(*hdr));      //Add taint T1 to hdr
4   ...
5   if ( hdr.size < MAXSIZE){                     //Branch controlled by T1
6     ...
7     msg = kmalloc(hdr.size, GFP_KERNEL);
8     if (!msg)
9       return -ENOMEM;
10    ...
11    copy_from_user(msg, uptr, hdr.size);        //Add taint T2 to msg
12    ...
13    process(msg);
14    ...
15    copy_to_user(msg, uptr, msg.size);          //Usage of double fetched value
16    ...
17  }
18  else
19    return -EINVAL;
20  ...
21 }
```

Fig. 7 Typical double-fetch bug case

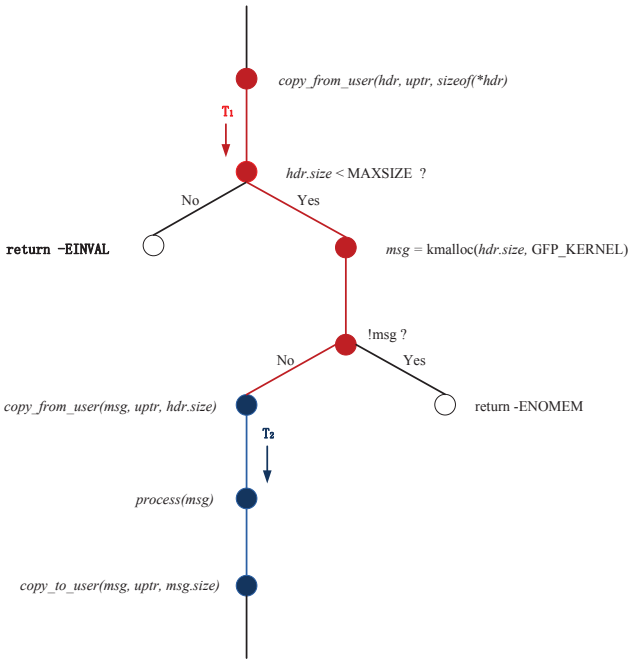


Fig. 8 Execution diagram of a conventional double-fetch bug case

return value of a function call (line 7). When the second fetch takes place, the new fetched value is tainted by new taint `T2` (line 11); therefore, `msg` carries both `T1` and `T2`. Finally, when `msg.size` is used (line 15), we check whether this occurs within a tainted branch controlled by a different taint that from the same memory location. In this example, the use of `msg.size` resides in two branches (line 5 and line 8), both of which are controlled by `T1`, but `msg.size` is also tainted by `T2`. Then, a double-fetch bug pattern is matched, and a potential double-fetch bug is indicated. A potential double-fetch bug is reported when there exists a situation in which the use of a tainted value occurs within a different taint controlled branch. The difference between the taints that come from the branch condition (*check*) and used value (*use*) indicate the potential inconsistency between the check and the use of the user data. Therefore, a potential double-fetch bug is indicated.

We can see this more clearly from the program execution diagram in Fig. 8. The red line indicates the execution paths affected by the first fetch, namely tainted by `T1`; whereas blue lines are paths affected by the second fetch, namely tainted by `T2`. As we can see, the `T1` controlled branch conditions steer the program execution, and the use of `msg.size` can occur only if the branch condition is satisfied, which depends on the first fetch. However, even if the first fetch makes this happen, the second fetch that takes place within the tainted branch is still free from constraints. Therefore, the fetched value can be any malicious value that destroys the kernel use, leading to serious consequences. However, if the used value and branch condition have the same taint, it indicates the same value is passed down from check to use, the consistency is guaranteed, and no problems will arise.

4 Implementation

We implemented a prototype based on our proposed approach named DFTracker, which works as a customized checker of the Clang Static Analyzer[1)] to automatically detect double-fetch bugs from programs when compiling the source code using Clang. Clang is a compiler frontend for the C language family, and it uses LLVM as its backend. The Static Analyzer is part of the Clang project and implemented as a C++ library that can be used by other tools.

The analyzer core performs symbolic execution of the given program by walking Clang's control flow graph (CFG). All the input values are represented by symbolic values; further, the engine deduces the values of all the expressions in the program based on the input symbols and paths. The execution is path-sensitive and every possible path through the program is explored. It collects the constraints on symbolic values along each path, and uses constraints to determine the feasibility of paths. As a regular user, we construct our own checker with the interfaces (callback functions) provided by the framework, regardless

[1)] http://clang-analyzer.llvm.org/checker_dev_manual.html

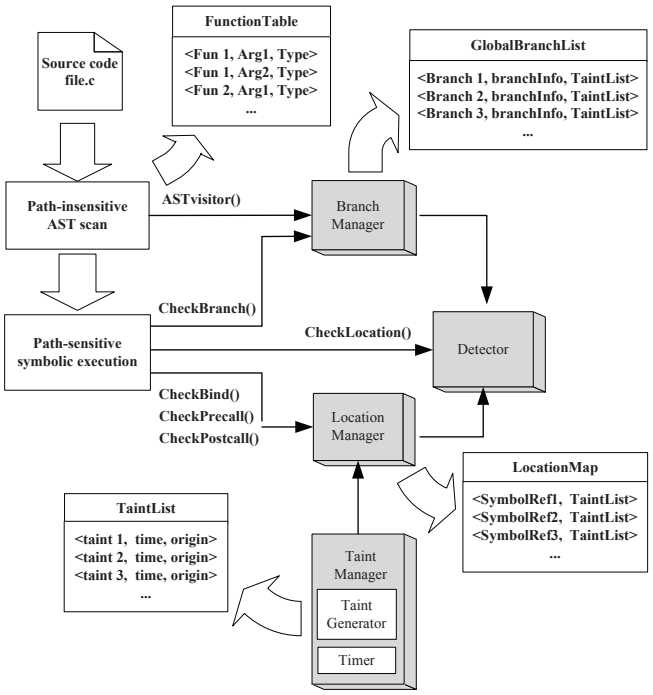


Fig. 9 Architecture of the DFTracker implementation

of how the core symbolic execution works. When the customized checker is invoked by the analyzer, the source code is analyzed automatically in a path-sensitive manner.

As shown in Fig. 9, DFTracker is mainly composed of four parts: Branch Manager, Location Manager, Taint Manager, and Detector. The whole detection procedure is divided into two phases: a path-insensitive abstract syntax tree (AST) scan and a path-sensitive symbolic execution.

4.1 Path-Insensitive AST Scan

The main goal of this procedure is to obtain useful information about the source code, such as the branch information and the function information. For each branch, we record its location and branch condition in a struct called `GlobalBranchList`, which is performed by Branch Manager. For the function definitions in the code, we record their `FunctionDecls` in a struct called `FunctionTable`, which includes the function names, function arguments, and arguments types. This procedure is very fast as it is only a quick scan of the AST, which is a preprocess of the symbolic execution in the second phase.

4.2 Path-Sensitive Symbolic Execution

In this phase, we conduct a full-path exploration of the tested program by symbolic execution. During the execution, DFTracker interacts with the Clang Static Analyzer mainly by the following callback functions provided by the infrastructure: `CheckLocation()`, `CheakBind()`, `CheckBranch()`, `CheckPreCall()`, `CheckPostCall()`. The components of DFTracker work in the following manner.

(1) Taint Manager Taint Manager deals with taints, which includes creating new taints, adding taints to a TaintList, and some other searching and checking operations that involve taints. In our system, a *Taint* struct is manifested as a 3-tuple (*number*, *time*, *origin*), which indicates the taint number, the timestamp when the taint is added, and the user memory location where this tainted value comes from. Taint Manager also creates a struct called *TaintList*, which is used to store multiple taints at one time. When we propagate taints from one variable to another or from a variable to a branch, it is the TaintList that is actually being passed. When a TaintList `tl1` is passed to a variable `a`, and `a` already has a TaintList called `tl0`, then these two TaintLists will be merged, and we keep the new merged TaintList. The merge process complies with the following rules.

- For taints from the same origin and have the same tag but with different timestamps, only the taint with the newest timestamp is retained, which means they are from the same kernel read but propagete with different speed in the kernel. Thus, a new taint will overwrite the old ones.
- For taints from different origins or the from the same origin but with different tags, all are retained.

(2) Location Manager Location Manager works on a struct called `LocationMap`, which maps the `SymbolRef` of each variable to a TaintList. As is well known, we are not able to obtain user data by pointer dereference directly in Linux kernel, and the only way to do this is through the so-called copy functions, such as `copy_from_user()` and `get_user()`. Therefore, the copy functions should be our start point. We hook the copy functions in the callback function `CheckPostCall()`, and add the new taint to the fetched value (the corresponding arguments) every time a copy function is invoked. Then, we pass taints to the variable that is assigned by a tainted variable, or to the return value of a function that has a tainted argument. The taint-passing procedure is conducted in callback functions `CheckBind()` and `CheckPostcall()`. This is how the taint propagation basically works in DFTracker.

(3) Branch Manager In addition to recording branch information to the `GlobalBranchList` in the path-insensitive phase, Branch Manager passes taints to

Algorithm of Double-Fetch Detection

```
Input: val - variable that being accessed
Input: gbl - GlobalBranchList that records all the branch info
Output: ret - retval that indicates whether it is a Double-Fetch
begin:
  if ( isTainted(val))
    v_taintList = get_taintList(val)
    if( gbl.findBranchByLoc(getSourceLoc(val))
      branch = gbl.getBranchByLoc(getSourceLoc(val)
      b_taintList = get_taintList(branch)
      for i in v_taintList :
        for j in b_taintList:
          if(i.taint != j.taint && i.origin == j.origin)
            return true
          end
        end
      end
    end
  end
  return false
end
```

Fig. 10 Algorithm of double-fetch detection

branches, which plays an important role in the propagation of taints in our approach. This is performed in the `CheckBranch()` callback function. When the symbolic execution reaches a branch, we examine the branch condition; if the condition expression involves tainted variables, then the branch should be tainted, and all the taints that come from this branch condition should be added to the TaintList associated with this branch. Here, we also need to merge all the TaintLists that come from different variables in the branch condition expression.

(4) Detector Detector is the brain of DFTracker, which decides whether a case should be reported as a potential double-fetch bug or not. With the help of the `CheckLocation()` callback function, every time a variable is accessed, we check whether this variable is tainted: if yes, we continue to check whether this happens within a branch controlled by a different taint that comes from the same origin. If yes, then a situation that the checked value and the used value come from different memory reads is indicated, and a potential double-fetch bug will be reported. The algorithm is shown in Fig. 10.

4.3 Optimization

To achieve better performance, we use the following strategies to accelerate the execution and lower the overhead.

(1) Coarse Timeline. The timestamp system in DFTracker is provided by a local timer. To lower the overhead, we use a coarse time mechanism, which means the timer only provides the relative orders of some critical events, rather than an accurate system timestamp. The timeline will be changed only when a taint is created, or a taint is passed to another variable. Other events that have nothing to do with the taint propagation will not affect the timeline.

(2) Skip functions without pointer argument. Kernel reads are necessary to make a double-fetch bug, and a kernel function will not fetch data without a user pointer, which means a pointer-type argument is essential for a kernel function to make a double-fetch bug. We record the function information in the AST scan phase, which includes the function names, function arguments, and argument types. If DFTracker obtains a function that does not have a pointer argument, it will just skip it without checking to accelerate the execution.

(3) Remove taint-free branches. In the AST scan phase, we record all the branches in a struct called `GlobalBranchList`. However, some of the branches will never be tainted, which are called taint-free branches, and we shall remove them from the `GlobalBranchList` in order to accelerate the execution when searching for a branch in the list. Every time we obtain a branch in the `CheckBranch()` callback function, we pass taints to the branch if the branch condition involves tainted variables. Otherwise, the branch is taint-free, and we remove it from the `GlobalBranchList` to accelerate the later check.

5 Evaluation

In this section, we provide the evaluation of our proposed approach. The evaluation is conducted from aspects of effectiveness, efficiency, and scalability. We also compare our approach with the work of Bochspwn over these aspects. Our test suite is composed of two parts, Benchmark Test and Real Case Test; see Table 1.

- Benchmark Test (No.1–No.5) focuses on the validity of the implementation of our proposed approach. The tests include the two basic double-fetch bug patterns, the propagation of the multi-taints, such as situations of multiple tainted variables, cases of multiple functions in a single file, and situations of double-fetch bugs happening across embedded functions. Some corner cases are also tested.
- Real Case Test (No.6–No.13) use cases of previous real-world double-fetch bugs, which include cases from

Table 1 Test Suite Introduction

No.	Name	Loc	Description
1	Pattern1	57	Basic Pattern 1 of double fetch in our proposed approach.
2	Pattern2	60	Basic Pattern 2 of double fetch in our proposed approach.
3	Multi_Taints	67	Test of multiple taints from multiple source parallel propagation.
4	Multi_Functions	87	Test of multiple taints parallel propagation in different functions.
5	Embedded_Calls	57	Test of multiple taints propagate across embedded function calls, double fetch happens across functions.
6	Win_Win32k	58	Real Windows double-fetch bug in win32k.sys [14].
7	Win_Memcmp	47	Real Windows double-fetch bug in memcmp() of Windows 8 [15].
8	Linux_Compat	598	Real double-fetch bug in compat.c of Linux-2.6.9 [35].
9	Linux_MicVirto	812	Real double-fetch bug in mic_virto.c of Linux-4.5 [36].
10	Linux_SclpCtl	145	Real double-fetch bug in sclp_ctl.c of Linux-4.5 [37].
11	Linux_CrosEcDev	353	Real double-fetch bug in cros_ec_dev.c of Linux-4.5 [38].
12	Linux_Auditsc	133	Real double-fetch bug in auditsc.c of Linux-4.5 [39].
13	Linux_Commctrl	100	Real double-fetch bug in commctrl.c of Linux-4.5 [40].

both Linux and Windows platform. To perform the comparison with the work of Bochspwn, we choose two real cases from Windows that were discovered by them.

All the experiments were conducted on a machine with a 1.4 GHz, quad-core CPU, 8 GB physical memory, running OSX 10.11.1 and Clang infrastructure version 3.8.0. The source code of DFTracker is about 2,000 lines of code (LOC).

5.1 Effectiveness

In this test, all the double-fetch bugs in the test suite were reported, which proves the effectiveness of DFTracker (shown in Table 2). In addition, no false negatives were reported, which benefits from the full-path exploration in the static approach. This is a big advantage over the widely used dynamic approaches such as Bochspwn [15], which usually produce large numbers of false negatives due to the poor path coverage.

Table 2 Effectiveness Test Result

No.	Name	Pattern	True Report	False Positives	False Negatives
1	Pattern1	1	1	0	0
2	Pattern2	2	1	0	0
3	Multi_Taints	1	1	0	0
4	Multi_Functions	1	2	0	0
5	Embedded_Calls	1	1	0	0
6	Win_Win32k	1	1	0	0
7	Win_Memcmp	1,2	2	2	0
8	Linux_Compat	1	1	2	0
9	Linux_MicVirto	1	1	0	0
10	Linux_SclpCtl	1	1	1	0
11	Linux_CrosEcDev	1	1	0	0
12	Linux_Auditsc	1	1	2	0
13	Linux_Commctrl	1	1	0	0

As for the false positives, technically, they are duplicate reports, which means a true bug is reported more than once, rather than a non-bug case being reported. This is because, based on our approach, DFTracker checks the potential double-fetch bug when the symbolic execution reaches the use of a tainted variable. However, once a double-fetch pattern is matched, any use of that tainted variable within a taint-controlled branch is reported. In other words, if the tainted variable is used more than once within a taint-controlled branch, each is reported. Therefore, the "false positives" in Table 2 are actually duplicate reports of the same double-fetch bug.

5.2 Efficiency

In the efficiency test, we paid attention to the overhead introduced by DFTracker. As listed in Table 3, we set execution baselines for comparison, and categorize the tests as native compile, DFTracker disabled, and DFTracker enabled. First, we natively compiled all the programs in the test suite with Clang: results are shown in column 3, which is the first baseline. Then we ran them again with Clang Static Analyzer but disabled all the checkers: results are shown in column 4, which is the second baseline. Finally, we ran tests with DFTracker enabled, and the results are shown in column 5. We calculated the overheads of DFTracker disabled compared with native compiling; the result is marked as `O1` in column 6. Then, we compared the

Table 3 Efficiency Test Result

No.	Name	Native Compile (s)	DFTracker Disabled (s)	DFTracker Enabled (s)	Overhead O_1 (%)	Overhead O_2(%)	Overhead O_2 - O_1(%)
1	Pattern1	0.277	0.418	0.436	84.1	92.1	8.0
2	Pattern2	0.196	0.408	0.424	108.2	116.3	8.1
3	Multi_Taints	0.194	0.422	0.429	117.5	121.1	3.6
4	Multi_Functions	0.211	0.419	0.454	98.6	115.2	16.6
5	Embedded_Calls	0.196	0.410	0.425	109.2	116.8	7.6
6	Win_Win32k	0.198	0.403	0.430	103.5	171.2	67.7
7	Win_Memcmp	0.189	0.407	0.813	115.3	330.2	215.0
8	Linux_Compat	0.233	0.635	0.949	172.5	307.3	134.8
9	Linux_MicVirto	0.217	0.533	0.872.	145.6	301.8	156.2
10	Linux_SclpCtl	0.205	0.482	0.753	135.1	267.3	135.2
11	Linux_CrosEcDev	0.198	0.443	0.704	123.7	255.6	131.9
12	Linux_Auditsc	0.262	0.750	1.280	186.3	388.5	202.2
13	Linux_Commctrl	0.245	0.698	1.255	184.9	412.2	227.3

overheads of DFTracker enabled with native compiling, and the result is marked as `O2` in column 7. Finally, the real overhead introduced by DFTracker is `O2 - O1`, which is shown in column 8.

From the results, we can see that the average overhead of DFTracker is 101.1%, and approximately 2X at most, which is acceptable for the single-file compiling. The overhead is much lower than the dynamic approach that Bochspwn adopted [15]. Theoretically, the overhead is mainly introduced by the symbolic execution phase: the more branches the tested program has, the longer time it takes. This explains why the execution times of the real cases, which are branch intensive, are much longer than the rest. In our future work, we will try to prune some infeasible paths to speed up the symbolic execution.

5.3 Scalability

DFTracker is implemented as a customized checker of the Clang Static Analyzer, which can be applied to any C/C++ source code that is compiled by Clang. In general, compiling the whole Linux kernel with Clang is not feasible because some features in the kernel are only supported by the GCC compiler, such as the `_asm{}` style of inline assembly. However, with the help of the LLVMLinux project[2)], we successfully compiled Linux kernel with Clang and tested DFTracker.

The test was conducted on an Ubuntu 14.04 machine with Linux kernel 3.18, which is a long-term support version. The compilation took approximately 18 h and 13 min, whereas a native compilation of the kernel without the Static Analyzer takes approximately 2 h. Therefore, the total overhead is approximately 9X, which is acceptable for a full-path exploration on the whole Linux kernel.

In total, we obtained 215 reports from the test. We used a Python script to filter out the duplicate reports as we mentioned in Section 5.1, and 61 cases remained. Then, we manually analyzed these cases and found a new double-fetch bug that was previously unknown in file `/fs/fhandle.c` (a detailed analysis is given in Section 6.1). As for the rest of the cases, five cases were double-fetch situations that were protected by the schemes as we describe in Section 6.3. Another 40 cases matched the double-fetch bug patterns, but the fetched value could not cause an error. Finally, 15 false reports were obtained due to the inaccuracy of the implementation.

5.4 Comparison

As the double-fetch bug is a relatively new topic, few systematic studies have been conducted except for the pilot work of Bochspwn. Therefore, only the work of Bochspwn is closely related to ours so that we could make a comparison. However, owing to the unavailability of the source code, we cannot duplicate the work directly. Hence, we make the comparison indirectly from the results of their experiments.

From the view of effectiveness, `Win_Win32k` and `Win_Memcmp` are the cases that could be found by Bochspwn and used as examples to demonstrate their work; our approach could find them as well. However, other Linux real cases such as `Linux_Compat` cannot be found by

[2)] http://llvm.linuxfoundation.org/

Bochspwn owing to its Windows-specific implementation, whereas our approach could successfully find them all.

In addition, DFTracker could find all the double-fetch bugs in the test suite with no false negatives and minor false positives (duplicate reports as we explained), whereas the dynamic approach adopted by Bochspwn usually produces large numbers of false negatives due to the poor path coverage.

As for the efficiency, DFTracker introduces overhead of approximately 2X for single files (real cases) and approximately 9X for the whole Linux kernel, which is acceptable. Bochspwn introduced severe runtime overhead by the simulator in their dynamic approach. It took 15 h to boot Windows in the simulator, and this did not cover full path exploration.

Finally, for the aspect of scalability, DFTracker can analyze the entire Linux kernel with full path coverage within an acceptable time. However, Bochspwn is Windows-specific and without full path coverage, scalability is limited.

```
166 static int handle_to_path(int mountdirfd, struct file_handle __user *ufh,
167                 struct path *path)
168 {
169     int retval = 0;
170     struct file_handle f_handle;
171     struct file_handle *handle = NULL;
        ...
182     if (copy_from_user(&f_handle, ufh, sizeof(struct file_handle))) {
183             retval = -EFAULT;
184             goto out_err;
185     }
186     if ((f_handle.handle_bytes > MAX_HANDLE_SZ) ||
187        (f_handle.handle_bytes == 0)) {
188             retval = -EINVAL;
189             goto out_err;
190     }
191     handle = kmalloc(sizeof(struct file_handle) + f_handle.handle_bytes,
192                      GFP_KERNEL);
193     if (!handle) {
194             retval = -ENOMEM;
195             goto out_err;
196     }
197     /* copy the full handle */
198     if (copy_from_user(handle, ufh,
199                        sizeof(struct file_handle) +
200                        f_handle.handle_bytes)) {
201             retval = -EFAULT;
202             goto out_handle;
203     }
204
205     retval = do_handle_to_path(mountdirfd, handle, path);
        ...
}
```

Fig. 11 New double-fetch bug in `fhandle.c`

6 Discussion

6.1 About the Bug

In the evaluation section, our approach successfully found all the already known real bugs in the first part and found a new double-fetch bug in the second part performed on Linux kernel-3.18. Here we provide a detailed analysis of the new double-fetch bug in `\fs\fhandle.c`.

As we can see from Fig. 11, there are two kernel data fetches in function `handle_to_path()` (line 182 and line 198), both via user pointer `ufh`. However, the first fetch only copies a struct header `f_handle` (tainted as `T1`) to check the real struct length from its element `f_handle.handle_bytes` (also tainted as `T1`) and allocate a kernel buffer `handle`. After the second fetch, the whole user data is copied to `handle`, which is tainted as `T2`. Then `handle` is used at line 205, which occurs within a branch (line 186) controlled by the first fetched value `f_handle.handle_bytes` (tainted as `T1`). Since `handle` also has the element `handle_bytes`, if this `handle_bytes` is used again later on, then it is a situation that matches the double-fetch bug pattern 1 as we proposed. Both fetches involve `handle_bytes`; the first one controls the branch (check), whereas the second one is for use, indicating a potential data inconsistency. This can cause a problem if this value is changed between the two fetches under a race condition.

To demonstrate how the problem is caused by this double-fetch bug, we tracked the tainted value during the propagation and demonstrated how it is used. At line 205 of Fig. 11, the tainted value `handle` is used as a parameter to function `do_handle_to_path()`. Then, within this function, as Fig. 12 shows, the taint propagates to `handle_dwords` (line 150), and then `handle_dwords` is used as a parameter in the invocation of function `exportfs_decode_fh()` (line 152). Within this function in file `expfs.c`, the tainted value `fh_len` is used both at line 426 and line 480, which are passed as parameters both to function `fh_to_dentry()` and `fh_to_parent()`. We then tracked the implementation of these two functions in file `export.c` and found that the tainted value `fh_len` is used for the verification of the struct length (line 243 and line 257). Malicious modification of this value will disrupt the verification here, causing an invalid message skip the check and be processed by the kernel as a normal one. Serious consequences such as over boundary access and buffer overflow will be caused.

6.2 Exploitation

A double-fetch bug can turn into a double-fetch vulnerability if the consequence is exploitable. As shown in

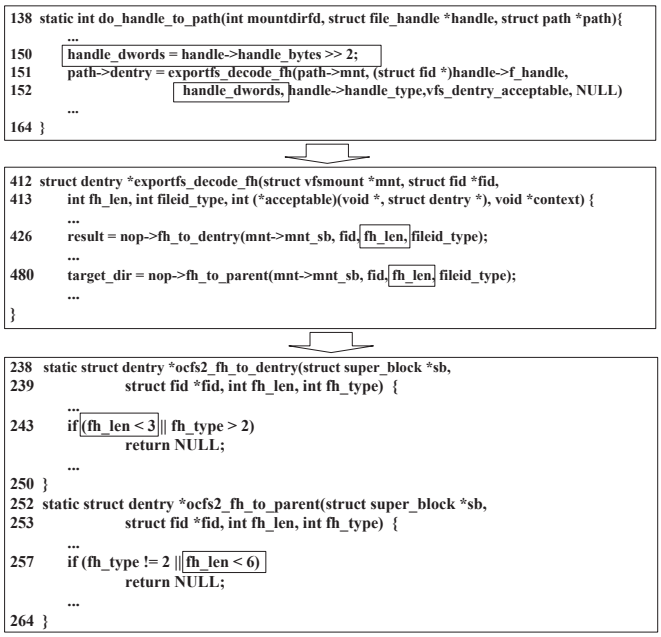


Fig. 12 How the second fetched value in `fhandle.c:` is used

Fig. 1, the exploitation works by changing the user data from a concurrently running user thread. This malicious user thread should be running in the same address space with the user thread that invokes the kernel function. Otherwise, it would be infeasible to rewrite the user data due to the address space isolation. This malicious thread should start right after the first fetch, continuously rewriting the specific memory location within the time window between the first and second fetches. Tricks such as using page boundary, disabling page cacheability, and translation lookaside buffers (TLB) flushing could significantly expand the time window between the two kernel reads to increase the success rate of changing the data [15]. The most critical part of the exploitation is finding the right memory location to rewrite the data. If a value that controls a loop or a copy length is changed, serious consequences such as buffer overflow, information leakage, and kernel crash would occur.

6.3 Double-Fetch Bug Prevention

Based on our study and analysis, we propose some advice on preventing double-fetch bugs.

1. **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 the developers only use the data from one of the fetches, problems can be avoided.
2. **Overwrite data**. There may be a double-fetch situation in which the first recommendation cannot be applied to resolve the situation, and some data need to be fetched and used twice. One way to resolve the situation is to overwrite the data from the second fetch with the data that have been fetched first. Even if a malicious user changed the data between the two fetches, the change would have no impact.
3. **Compare data**. Another way to resolve a double-fetch situation is not to overwrite the data, but to compare the data from the first fetch with the data of the second fetch. If the data are not the same, the operation can be aborted safely. In this situation, a double fetch is not avoided, but an attack can be identified.
4. **Synchronization approach**. The last way to prevent a double-fetch bug is using synchronization approaches to keep the atomicity of the 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 no problems will arise. Even though synchronizations could prevent potential race conditions, this approach will inevitably cause performance degradation for the kernel.

6.4 Limitations

Even though DFTracker can effectively find double-fetch bugs, as an innovative prototype, it still has some limitations. For example, DFTracker can only detect double-fetch bugs in C/C++ programs with open source code. For the patterns we proposed, we currently do not consider the situations that the second kernel read and the use of the fetched value is not explicitly guarded by a branch, because in this situation, the branch controlled by the first kernel read does not affect the execution of the second kernel read, and the use of the second fetched value is not based on the check of the first fetched value. Therefore, most of the cases in this situation are not causing a double-fetch bug. However, this decision could miss situations where the branch jumps to the error-handling code or just returns, which could affect the execution of the second kernel read, causing a double-fetch bug. This could cause false negatives and should be tackled in future work. In addition, as we mentioned in Section 5.1, multiple uses of the tainted variable in a same double-fetch case will result in duplicate reports, which should be eliminated in future work.

7 Related Work

A double-fetch bug is a type of special race condition between the kernel and the user space. To date, little work has been conducted on double-fetch bugs owing to their uncertainty in appearance and difficulty in digging from the kernel.

The Bochspwn project [15] is the only systematic work on double-fetch bugs presented so far, and was the first to formally detect it. Their research has been significant in finding double-fetch bugs based on memory access pattern. However, there still could be some improvements in accuracy and efficiency, such as eliminating the false positives and false negatives. Their work was conducted on a Windows platform with a dynamic approach, and our work is performed on a Linux platform with a static approach, which complements their work well. To the best of the authors' knowledge, the work presented in this paper is the first to detect double-fetch bugs with a static approach, and the static approach can detect bugs without actual executions, which is much faster than dynamic approaches. Besides, the static approach has better path coverage.

Yang's work [30] is very relevant to ours. They focused on concurrency bugs in a TOCTOU situation, and they believed such bugs could be exploited to carry out concurrency attacks. They also studied some real cases to catalog concurrency attacks and pointed out that the risk of concurrency attacks is proportional to the duration of the vulnerability window. However, they did not propose an actual solution of how to find or prevent such bugs, except for some implications.

DataCollider [41] is a lightweight tool that detects data races in kernel modules. To reduce the runtime overhead, DataCollider randomly samples a small percentage of memory accesses as candidates for data-race detection. It uses breakpoint facilities already supported by hardware architectures to lower the runtime overhead, and it is also oblivious to the synchronization protocols. However, it only concentrates on the races happening within the kernels. As a type of dynamic approach, it has to run the program multiple times to improve the path coverage as well as trigger the bug. In addition, even though the sampling technique in DataCollider lowers the runtime overhead, it reduces the chances of finding a bug.

SKI [42] also detects concurrency bugs in the kernel. It explores the kernel interleaving space in a systematic way by taking full control over the kernel thread interleavings. To control the thread interleavings without modifying the kernel code, SKI uses an adapted virtual machine monitor that determines the status of the various threads of execution and selectively blocks a subset of these threads to enforce the desired schedule. However, a dynamic approach that involves thread interleavings will inevitably cause false negatives. In addition, runtime overhead will increase dramatically as the threads increase.

Engler et al. used both static analysis and software model checking [43] to find software errors [44]. ARCHER [45] is a static memory access checker that uses path-sensitive, inter-procedural symbolic analysis to bound the values of both variables and memory sizes. It evaluates known values using a constraint solver at every array access, pointer dereference, or call to a function that expects a size parameter. Accesses that violate constraints are flagged as errors. This checker used a similar technique to our approach except that its checking is not specific to the double-fetch bug. RacerX [22] is a static tool that uses flow-sensitive, inter-procedural analysis to detect both race conditions and deadlocks. It uses novel techniques to counter the impact of analysis mistakes, and tracks a set of code features that it uses to sort errors from most to least severe. However, this tool does not involve the special situation of racing between the kernel and user. In addition, they invented extensible languages for program analysis.

8 Conclusion

In this work, we focused on detecting double-fetch bugs between the kernel and the user space. We proposed an innovative approach of multi-taint parallel tracking. To the best of our knowledge, we are the first to introduce multiple-taint parallel tracking into double-fetch bug detection. There are two phases in our proposed approach. First, a quick scan on the AST is conducted to collect source code information, which includes the branch and function information. Then, a path-sensitive symbolic execution is followed to explore all the paths to detect double-fetch bugs based on our proposed patterns. We implemented a prototype called DFTracker based on our proposed approach, and its viability was proved by our experiments. All the double-fetch bugs in the test suite were detected with minor false positives and no false negatives. We tested DFTracker on the whole Linux kernel and found a new double-fetch bug. The average overhead of DFTracker is 2x

for single files and 9x for the entire Linux kernel test, which is acceptable.

Our proposed approach provides a new perspective for double-fetch bug detection—specific to double-fetch bug features and has better path coverage and lower runtime overhead—which is more suitable for in-house testing before the software release. In future work, we will try to improve the performance by combining our work with techniques such as program slicing [46] and parallel execution [13].

9 Acknowledgments

The authors thank the anonymous reviewers for their helpful feedback. The work is supported by The National Key Research and Development Program of China (No. 2016YFB0200401).

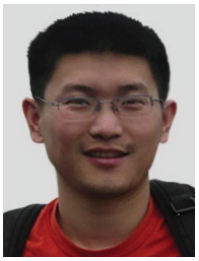
References

1. Leveson N G, Turner C S. An investigation of the therac-25 accidents. Computer, 1993, 26(7): 18–41
2. Jesdanun A. General electric acknowledges northeastern blackout bug, 20ww
3. Net X. Nasdaq ceo blames software design for delayed facebook trading, 2012
4. Kasikci B, Zamfir C, Candea G. Data races vs. data race bugs: telling the difference with portend. ACM SIGPLAN Notices, 2012, 47(4): 185–198
5. Huang J, Meredith P O, Rosu G. Maximal sound predictive race detection with control flow abstraction. ACM SIGPLAN Notices, 2014, 49(6): 337–348
6. Narayanasamy S, Wang Z, Tigani J, Edwards A, Calder B. Automatically classifying benign and harmful data races using replay analysis. In: ACM SIGPLAN Notices. 2007, 22–31
7. Dimitrov D, Raychev V, Vechev M, Koskinen E. Commutativity race detection. In: ACM SIGPLAN Notices. 2014, 305–315
8. Cai X, Gui Y, Johnson R. Exploiting unix file-system races via algorithmic complexity attacks. In: Security and Privacy, 2009 30th IEEE Symposium on. 2009, 27–41
9. Hsiao C H, Yu J, Narayanasamy S, Kong Z, Pereira C L, Pokam G A, Chen P M, Flinn J. Race detection for event-driven mobile applications. In: ACM SIGPLAN Notices. 2014, 326–336
10. Maiya P, Kanade A, Majumdar R. Race detection for android applications. In: ACM SIGPLAN Notices. 2014, 316–325
11. ChinaByte . Amazon ec2 reboot to cope with xen vulnerability, 20ww
12. Gunawi H S, Hao M, Leesatapornwongsa T, Patana-anake T, Do T, Adityatama J, Eliazar K J, Laksono A, Lukman J F, Martin V, others . What bugs live in the cloud? a study of 3000+ issues in cloud systems, 2014
13. Wu Z, Lu K, Wang X, Zhou X, Chen C. Detecting harmful data races through parallel verification. The Journal of Supercomputing, 2015, 71(8): 2922–2943
14. Serna F J. Ms08-061: The case of the kernel mode double-fetch, 2008
15. Jurczyk M, Coldwind G, others . Identifying and exploiting windows kernel race conditions via memory access patterns. 2013
16. Eckelmann S. [patch-resend] backports: Fix double fetch in hlist_for_each_entry*_rcu, 2014
17. Wilhelm F. Tracing privileged memory accesses to discover software vulnerabilities. Master's thesis, Karlsruher Institut für Technologie, 2015
18. Voung J W, Jhala R, 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, 205–214
19. Pratikakis P, Foster J S, Hicks M. Locksmith: Practical static race detection for c. ACM Transactions on Programming Languages and Systems (TOPLAS), 2011, 33(1): 3
20. Huang J, Zhang C. Persuasive prediction of concurrency access anomalies. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. 2011, 144–154
21. Chen J, 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, 8
22. Engler D, Ashcraft K. Racerx: effective, static detection of race conditions and deadlocks. In: ACM SIGOPS Operating Systems Review. 2003, 237–252
23. Sen K. Race directed random testing of concurrent programs. ACM SIGPLAN Notices, 2008, 43(6): 11–21
24. Kasikci B, Zamfir C, Candea G. Racemob: crowdsourced data race detection. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles. 2013, 406–422
25. Zhang W, Sun C, Lu S. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In: ACM SIGARCH Computer Architecture News. 2010, 179–192
26. Zhang W, Lim J, Olichandran R, Scherpelz J, Jin G, Lu S, Reps T. Conseq: detecting concurrency bugs through sequential errors. In: ACM SIGPLAN Notices. 2011, 251–264
27. Yu J, Narayanasamy S, Pereira C, Pokam G. Maple: a coverage-driven testing tool for multithreaded programs. In: Acm Sigplan Notices. 2012, 485–502
28. Bishop M, Dilger M, others . Checking for race conditions in file accesses. Computing systems, 1996, 2(2): 131–152
29. Watson R N. Exploiting concurrency vulnerabilities in system call wrappers. In: First USENIX Workshop on Offensive Technologies, WOOT '07. 2007
30. Yang J, Cui A, Stolfo S, Sethumadhavan S. Concurrency attacks. In: Presented as part of the 4th USENIX Workshop on Hot Topics in Par-

allelism. 2012
31. Chen H, Wagner D. Mops: an infrastructure for examining security properties of software. In: Proceedings of the 9th ACM conference on Computer and communications security. 2002, 235–244
32. Cowan C, Beattie S, Wright C, Kroah-Hartman G. Raceguard: Kernel protection from temporary file race vulnerabilities. In: USENIX Security Symposium. 2001, 165–176
33. Lhee K S, Chapin S J. Detection of file-based race conditions. International Journal of Information Security, 2005, 4(1-2): 105–119
34. Payer M, Gross T R. Protecting applications against tocttou races by user-space caching of file metadata. In: ACM SIGPLAN Notices. 2012, 215–226
35. Cox M J. Bug 166248 - can-2005-2490 sendmsg compat stack overflow, 2005
36. Wang P. Double-fetch bug in drivers/misc/mic/host/mic_virtio.c of linux-4.5, 2016
37. Wang P. Double-fetch bug in drivers/s390/char/sclp_ctl.c of linux-4.5, 2016
38. Wang P. Double-fetch bug in drivers/platform/chrome/cros_ec_dev.c of linux-4.6, 2016
39. Wang P. Double-fetch bug in kernel/auditsc.c of linux-4.6, 2016
40. Wang P. Double-fetch bug in drivers/scsi/aacraid/commctrl.c of linux-4.5, 2016
41. Erickson J, Musuvathi M, Burckhardt S, Olynyk K. Effective data-race detection for the kernel. In: OSDI. 2010, 1–16
42. Fonseca P, Rodrigues R, Brandenburg B B. Ski: exposing kernel concurrency bugs through systematic schedule exploration. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014, 415–431
43. Yang J, Twohey P, Engler D, Musuvathi M. Using model checking to find serious file system errors. ACM Transactions on Computer Systems (TOCS), 2006, 24(4): 393–423
44. Engler D, Musuvathi M. Static analysis versus software model checking for bug finding. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. 2004, 191–210
45. Xie Y, Chou A, Engler D. Archer: using symbolic, path-sensitive analysis to detect memory access errors. ACM SIGSOFT Software Engineering Notes, 2003, 28(5): 327–336
46. Wu Z, Lu K, Wang X, Zhou X. Collaborative technique for concurrency bug detection. International Journal of Parallel Programming, 2015, 43(2): 260–285

Pengfei Wang received B.S. and M.S. degrees in 2011 and 2013, respectively, from the College of Computer, National University of Defense Technology, Changsha, China. He is now pursuing his Ph.D. in the College of Computer, National University of Defense Technology. His research interests include operating systems and software testing.

Kai Lu received a B.S. degree and Ph.D. in 1995 and 1999, respectively, from the College of Computer, National University of Defense Technology, Changsha, China. He is now a Professor in the College of Computer, National University of Defense Technology. His research interests include operating systems, parallel computing, and security.



Gen Li received a B.S. degree and Ph.D. in 2004 and 2010, respectively, from the College of Computer, National University of Defense Technology, Changsha, China. He is now an Assistant Professor in the College of Computer, National University of Defense Technology. His research interests include operating systems and software testing.

Xu Zhou received B.S. and M.S. degrees and a Ph.D. in 2007, 2009, and 2014, respectively, from the College of Computer, National University of Defense Technology, Changsha, China. He is now an Assistant Professor in the College of Computer, National University of Defense Technology. His research interests include operating systems and parallel computing.