DFTinker: Detecting and Fixing Double-fetch Bugs in an Automated Way

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

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

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

1 Introduction

The wide use of multi-core hardware is making concurrent programs increasingly pervasive, especially in operating systems, network systems, and even IoT devices. However, the reliability of such system is severely threatened by the notorious concurrency bugs [5,12]. Among all the concurrency bugs, the double-fetch bug is one of the most special and significant types.

Previous research focuses on the detection of double-fetch bugs. Dynamic approaches [3,10] detect double-fetch bugs by tracing memory accesses. However, such approaches are limited by the path coverage. They cannot be applied to code that needs corresponding hardware to be executed, so device drivers cannot be analyzed without access to the device or a simulation of it. Static approaches detect double-fetch bugs based on the identification of transfer functions [9,11], however, the accuracy and efficiency of such approaches are undesirable as they lack runtime information and still rely on manual efforts to confirm the bug. In addition, none of the previous works provides a practical solution on automatically fixing double-fetch bugs except some prevention suggestions. Thus, the fix of double-fetch bugs still relies on manually locating and rewriting the source code, and an automatic solution is in urgent need.

This paper proposes a comprehensive approach to automatically detect and fix double-fetch bugs. In the first phase, a static pattern-matching method based

* Corresponding author is Y. Luo (email: nudtlyq@163.com)

on the Coccinelle engine is used to identify double-fetch bugs. In the second phase, the identified bug is automatically fixed based on the support of the Intel Transactional Synchronization Extension (TSX). In summary, the main contribution of this paper is as follows:

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

2 Background

In modern operating systems, the kernel space is always separated from the user space for safety [8]. Kernel code run in the kernel space and get data from users if needed, it will use specific functions, termed *transfer functions*. In Linux kernel, there are four typical transfer functions, `get_user()`, `put_user()`, `copy_from_user()`, `copy_to_user()`. All their effects are fetching data or transferring data between the kernel space and the user space. Malicious changes between two fetches many cause fatal errors in kernels, termed double-fetch bugs.

Coccinelle [7] 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. One of the advantages of Coccinelle engine is path-sensitive, it is specially optimized for traversing paths.

Traditionally, transactional memory [1, 2] is used to simplify concurrent programming. It allows executing load and store instructions in an atomic way. Transactional memory systems provide high-level instructions to developers so as to avoid low-level coding, and this achieves a better access model to shared memory in concurrent programming. Hardware transactional memory achieves transactions by processors, caches, and bus protocol. It provides opportunities to implement dynamic schedules according to specific CPU instructions. We choose Intel TSX to ensure data consistency.

3 Design

3.1 Detection of Double-fetch Bugs

As section 2 states, double-fetch bugs and transfer functions are closely related, and each fetch indicates an invocation of a transfer function. However, since there are many complex situations in the kernel code, such as pointer change and aliasing, double-fetch bug detection needs further and thorough analysis.

Table 1. Expanded Transfer Functions.

No.	Name	Type	Parameter
1	unsafe_get_user	macro	dst, src, err
2	__copy_in_user	macro	des, src, len
3	__copy_user	function	dst, src, len
4	__copy_user_zeroing	function	dst, src, len
...			

Wang *et al.* and Xu *et al.* all focus on four transfer functions to detect double-fetch bugs, i.e., `get_user()`, `__get_user()`, `copy_from_user()`, and `__copy_from_user()`, the functionality of which is transferring data from user space to kernel space.

However, these rules are not strong enough to cover all double-fetch bugs. We improve the rules 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 other functions containing transfer functions, and their targets are transferring data from user space to kernel space as well, such as `memdup_user()` mentioned above. Table 1 shows 4 of 15 functions (and macros) we include to detect double-fetch bugs.

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. Many cases are missed because of careless implementation. We fixed these rules 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. The first situation is when 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, that means, the second fetch will never be executed if the first fetch is executed. Thus, this is not a double-fetch bug actually. There are many cases like this in the kernel code.

2. Two fetches are in the different branches. Another situation is when the two fetches are in the different branches, just like a `SWITCH` statement. These conditions can never be satisfied at the same time, so this situation is also a non-double-fetch bug situation.

3.2 Automated Fixing with Intel TSX

Previous research only proposed suggestions on preventing double-fetch bugs [9, 11], such as don't copy the header twice. However, we need a practical solution to automatically fix the bug.

We implement fixing function with 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 transaction.

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

4 Implementation

DFTinker consists of three parts, a detector, a patcher and a supervisor. The detector is used for detecting double-fetch bugs. It is implemented as SmPL files and the Coccinelle engine. The Coccinelle engine will use these SmPL files as pattern rules to find double-fetch bugs and filter out non-double-fetch bugs cases. The patcher is used for fixing the double-fetch bugs, it is made up of header files and SmPL files. Header files are used for providing prevention interfaces and SmPL files are used for providing rules for fixing. The last part is the supervisor, which consists of Linux shell scripts. The supervisor is used for supervising the Coccinelle engine and sorting out the results of the experiments, so as to leave the process fully automated.

5 Evaluation

5.1 Detection of Double-fetch Bugs

The experiments are conducted on a Linux laptop running Ubuntu 16.04 x64, with one Intel i7-7700HQ 2.6GHz processor, 8GB of memory, 250GB SSD. We use Linux 4.14.10, OpenBSD 6.2, FreeBSD 11.1, Android 7.0.0 (kernel version 3.18), and Darwin 10.13.3, which were the relatively newer version when the experiments were conducted.

DFTinker is applied to the five prevalent open source kernels. The statistical result[1] is shown in Table 2. We find 24 double-fetch bugs in the Linux kernel, 12 bugs in the FreeBSD kernel, 41 bugs in the Android kernel, 4 bugs in the Darwin kernel. Note that DFTinker can identify all known double-fetch bugs in the Linux kernel including Wang *et al.*'s work and Xu *et al.*'s work, however, our approach is more concise in contrast, which proves DFTinker's efficiency.

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

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

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

5.2 Automated Fixing with Intel TSX

In our experiments, we fix target functions using DFTinker and modify the data between two fetches using a user thread. The results show that the data fetched at the second time is same as the first time, which proves that DFTinker is effective in protecting double-fetch bugs.

6 Discussion

Jurczyk and Coldwind [3,4] used a dynamic approach in their Bochspwn project to study double-fetch bugs in Windows. By tracing memory accesses, they successfully found double-fetch bugs in the Windows kernel. However, such dynamic approaches can not test code under strict conditions. Our static approach has a better code coverage and can detect double-fetch bugs in the drivers, where the dynamic approaches are incapable of.

Schwarz *et al.* [6] proposed a method using cache-attack and kernel-fuzzing techniques to detect, exploit, and eliminate double-fetch bugs in Linux syscalls. However, their approach is limited to Linux syscalls, whereas large numbers of the double-fetch bugs occur in non-syscall functions, such as functions in drivers, are missed. Thus, their approach suffers from a low code coverage, whereas our approach is free from that. As for DropIt they implement, our approach can fix double-fetch bugs automatically instead of manual fixing.

Xu *et al.* [11] proposed a formal definition of double-fetch bugs and used a static analysis based on LLVM IR and symbolic execution to detect such bugs. However, their definition takes all the potential situations into consideration, which are not currently buggy but only have the potential to turn into bugs when the code is updated. Besides, their approach needs to compile the source code to LLVM IR and specify the target architecture. Thus, it detects only one architecture at one time, leading to the miss of bugs such as CVE-2016-6130. Our approach has a better code coverage, which can analyze the source code of all the architecture at one time.

Although DFTinker achieves a decent performance in detecting and fixing double-fetch bugs, it relies on the availability of the source code, which is suitable for in-house testing. We will take the binary situations into consideration in the future work.

7 Conclusion

This paper proposes an approach to automatically detect and fix double-fetch bugs. We implement a prototype named DFTinker and evaluate it with real kernels. Experiments show that DFTinker is effective and efficient in automatically detecting and fixing double-fetch bugs. DFTinker detected 81 cases in prevalent kernels and succeed in defending malicious data tampering.

8 Acknowledgements

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

References

1. 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.
2. M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. pages 289–300, 1993.
3. 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`.
4. 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`.
5. X. Ma, Y. Wang, Q. Qiu, W. Sun, and X. Pei. Scalable and elastic event matching for attribute-based publish/subscribe systems. *Future Generation Computer Systems*, 36(7):102–119, 2014.
6. 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.
7. H. Stuart. Hunting bugs with coccinelle. *Masters Thesis*, 2008.
8. 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.
9. 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.
10. F. Wilhelm. Tracing privileged memory accesses to discover software vulnerabilities. Master's thesis, Karlsruher Institut für Technologie, 2015.
11. 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`.
12. L. Yang, L. Yang, Y. Wang, B. Zhang, L. Ma, L. Ma, and X. Luo. Rule-based security capabilities matching for web services. *Wireless Personal Communications*, 73(4):1349–1367, 2013.