Static Race Detection in OS Kernels by Mining Locking Rules

Abstract

To take advantage of multiple-core architecture of modern CPUs, an operating system (OS) is often designed to be highly concurrent, and processes shared data structures in parallel. To ensure safe access to shared data structures, fine-grained locking mechanisms are used to avoid data races among different threads. However, it is hard to determine whether a lock should be used for a specific data structure field (locking rules) even for an expert developer, due to poor documentation and complicated logic of OS code. As a result, OS kernel is prone to data races, because necessary locks may be missed by mistake. Static analysis is a common technique to help improve code quality, but it is quite challenging to detect data races in OS kernels automatically, because of lack of knowledge of locking rules and high complexity of concurrent execution.

In this paper, we design a practical static analysis approach named RaceMiner, to effectively detect harmful races that can trigger memory or logical bugs in OS kernels by mining locking rules. RaceMiner first employs an alias-aware rule mining method to automatically deduce locking rules, and detects data races caused by violation of these rules. And then performs a lock-usage analysis to filter out false positives caused by concurrency. At last, RaceMiner extracts harmful data races from all detected data races through a pattern-based estimation. We have evaluated RaceMiner on Linux 6.2, and find xxx data races, with a false positive rate of xxx. Among these data races, xxx are estimated to be harmful. We have reported these harmful bugs to Linux kernel developers, and xxx of them have been confirmed.

1. Introduction

To take advantage of multiple-core architecture of modern CPUs, an OS kernel is often designed to be highly concurrent. However, concurrent execution can inevitably introduce concurrency bugs. Data race is a common kind of concurrency bugs that occurs when multiple threads access the same variable concurrently and at least one of the accesses is a write. Some data races are quite dangerous and can cause serious bugs such as null-pointer dereferences, data inconsistency and infinite loops.

To detect data races effectively, some approaches [1, 2, 3, 4, 8, 10, 11, 23, 27] rely on developers to supply annotations that describe the locking discipline such as which lock protects a specific data structure field, and detect data races by exploiting data structure field accesses that violate these disciplines. For example, Clang thread safety analysis [8] requires developers to label a variable and the lock to protect it with a `GUARDED_BY` attribute. And then detects data races by finding variable accesses that violate the given attribute. However, such annotation-based approaches are not suitable for race detection in OS kernels, because even an expert developer can not provide accurate annotations, due to poor documentation and complicated logic of OS codes.

Some static approaches [6, 9, 21, 22, 24] employ lockset-based analysis to detect data races automatically. They first collect locks that are acquired when a given variable is accessed, and then mine locking rules about whether a given variable should be protected by a specific lock. At last, they detect data races by checking whether variable accesses violate the mined locking rules. However, they do not consider alias relations [9, 24] or just use imprecise flow-insensitive alias analysis [6, 21, 22], and thus can introduce both false positives and false negatives.

Dynamic analysis can acquire run-time information such as memory address, and thus complex alias analysis can be avoided. Therefore, some approaches [13, 17, 18, 19, 20] detect data races dynamically to improve precision. They mine implicit code rules in softwares by statistically analyzing execution traces, and then use the mined rules to detect related bugs. For example, LockDoc [18] records accesses to data structure fields and lock acquisitions of an instrumented Linux kernel, and then infers locking rules from the recorded accesses. After that, LockDoc automatically locates data structure fields access that violate the inferred locking rules to detect concurrency bugs such as data races. However, dynamic analysis suffers from low code coverage, and thus the locking rules inferred through execution traces can be imprecise. Besides, dynamic analyses are hard to deploy, because they need to run OS kernels with instruments. At last, it is difficult for dynamic tools to estimate the harmfulness of a data race, because race conditions are hard to trigger by running the checked software [5, 12, 16, 26].

In this paper, we design a practical static analysis approach named RaceMiner, to automatically detect data races and estimate the harmfulness of these data races in OS kernels effectively. RaceMiner consists of three key techniques:

(T1) RaceMiner uses an *alias-aware rule mining method* to automatically deduce locking rules. We observe that an access to a data structure field is often protected by the lock in the same data structure as the accessed field. And this relationship between accessed field and the protecting lock can be effectively inferred from a special data structure named alias graph [14, 15]. Specifically, fields in the same data structure have the same ancestor in the alias graph, and thus whether an accessed field and a specific lock are in the same data structure

can be inferred by finding a common ancestor in the alias graph. Moreover, with benefits from precise field-sensitive alias relationships of alias graph, RaceMiner can accurately extract the accessed field and its corresponding protecting lock. After obtaining the accessed field and protecting lock pairs, RaceMiner calculates the proportion of field accesses protected by a specific lock in all field accesses. If the proportion is larger than a given threshold, the accessed field is determined to be protected by the protecting lock.

(T2) RaceMiner uses a *lock-usage analysis* to filter out false positives caused by concurrency. After mining locking rules, RaceMinder detects data races by checking whether a given data structure field access violates the locking rules. To reduce false negatives, RaceMinder conservatively assumes every two functions can be executed concurrently, and thus can introduce false positives. However, we observe that each kernel module has an initialization phase, after which many functions can be called concurrently by other modules. When performing initialization, the kernel module serially initialized the lock and prepare other data for subsequent operations. And thus functions with the lock initialization and functions called by them tend not to be executed concurrently. Based on this observation, RaceMiner extracts all functions reachable from the function with a lock initialization operation such as `spin_lock_init()`, and suppose that this function can not be executed in parallel.

(T3) RaceMiner uses a *pattern-based estimation* to extract harmful races that can trigger memory or logical bugs such as null-pointer dereferences and data inconsistency. Some data races are benign, and developers do not put effort into repairing them because they can not cause serious memory or logical bugs. Therefore, it is important to estimate the harm a data race can bring. We observe that harmful data races can be estimated by given patterns. For example, if a raced data performs as an operand of a dereference instruction, a null-pointer dereference can occur; if a raced data is accessed for more than once and data inconsistency can occur. Based on this observation, RaceMiner uses some patterns to extract harmful races that can cause memory or logical bugs automatically.

We have implemented RaceMiner with LLVM [7] and Z3 [25]. RaceMiner performs automated static analysis on the LLVM bytecode of the checked OS kernel. Overall, we make three main contributions in this paper:

- We analyze the challenges of data race detection in OS kernels, and propose three key techniques to address these challenges: (T1) an *alias-aware rule mining method* to automatically deduce locking rules; (T2) a *lock-usage analysis* to filter out false positives caused by concurrency; (T3) a *pattern-based estimation* to extract harmful data races that can trigger memory or logical bugs such as null-pointer dereferences and data inconsistency.
- Based on these three key techniques, we design a practical static analysis approach named RaceMiner, to effectively detect data races and estimate the harmfulness of these data races in OS kernels.
- We have evaluated RaceMiner on Linux 6.2, and find xxx data races, with a false positive rate of xxx. Among these data races, xxx are estimated to be harmful. We have reported these harmful bugs to Linux kernel developers, and xxx of them have been confirmed.

The rest of this paper is organized as follows. Section 2 introduces the motivation and challenges of data race detection in OS kernels. Section 3 introduces our key techniques to address these challenges. Section 4 introduces RaceMiner. Section 5 shows our evaluation. Section 6 makes a discussion about RaceMiner. Section 7 presents related work, and Section 8 concludes this paper.

2. Motivation

3. Race Detection by Mining Locking Rules

4. Framework

5. Evaluation

6. Discussion

7. Related Work

8. Conclusion

References

[1] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. SharC: checking data sharing strategies for multithreaded C. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 149–158, 2008. https://doi.org/10.1145/1379022.1375600.

[2] Zachary R Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 98–109, 2009. https://doi.org/10.1145/1542476.1542488.

[3] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2(OOPSLA):1–28, 2018. https://doi.org/10.1145/3276514.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 211–230, 2002. https://doi.org/10.1145/582419.582440.

[5] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010. https://doi.org/10.1145/1736020.1736040.

[6] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269, 2002. https://doi.org/10.1145/512529.512560.

[7] Clang: an LLVM-based C/C++ compiler, 2021. http://clang.llvm.org/.

[8] A C++ language extension which warns about potential race conditions in code, 2021. https://clang.llvm.org/docs/ThreadSafetyAnalysis.html.

[9] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003. https://doi.org/10.1145/945445.945468.

[10] Cormac Flanagan and Stephen N Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, 2000. https://doi.org/10.1145/349299.349328.

[11] Cormac Flanagan and Stephen N Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 90–96, 2001. https://doi.org/10.1145/379605.379687.

[12] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 221–230. IEEE, 2010. https://doi.org/10.1109/DSN.2010.5544315.

[13] Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded java programs. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 288–296. IEEE, 2008. https://doi.org/10.1109/ASE.2008.39.

[14] George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. An efficient data structure for must-alias analysis. In *Proceedings of the 27th International Conference on Compiler Construction (CC)*, pages 48–58, 2018. https://doi.org/10.1145/3178372.3179519.

[15] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 859–872, 2022. https://doi.org/10.1145/3503222.3507770.

[16] Peng Liu, Omer Tripp, and Charles Zhang. Grail: Context-aware fixing of concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 318–329, 2014. https://doi.org/10.1145/2635868.2635881.

[17] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. Wids checker: combating bugs in distributed systems. In *Proceedings of the 4th USENIX conference on Networked Systems Design and Implementation (NSDI)*, pages 19–19, 2007. https://dl.acm.org/doi/abs/10.5555/1973430.1973449.

[18] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. LockDoc: Trace-based analysis of locking in the Linux kernel. In *Proceedings of the 14th EuroSys Conference 2019*, pages 1–15, 2019. https://doi.org/10.1145/3302424.3303948.

[19] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 26th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 3–14, 2018. https://doi.org/10.1145/3236024.3236071.

[20] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A Popa, and Yuanyuan Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 103–116, 2007. https://doi.org/10.1145/1294261.1294272.

[21] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319, 2006. https://doi.org/10.1145/1133981.1134018.

[22] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331, 2006. https://doi.org/10.1145/1133981.1134019.

[23] Caitlin Sadowski and Jaeheon Yi. How developers use data race detection tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, pages 43–51, 2014. https://doi.org/10.1145/2688204.2688205.

[24] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 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 (ESEC/FSE)*, pages 205–214, 2007. https://doi.org/10.1145/1287624.1287654.

[25] Z3: a theorem prover, 2021. https://github.com/Z3Prover/z3.

[26] Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. Predicting concurrency bugs: how many, what kind and where are they? In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering EASE*, pages 1–10, 2015. https://doi.org/10.1145/2745802.2745807.

[27] Diyu Zhou and Yuval Tamir. PUSh: Data race detection based on hardware-supported prevention of unintended sharing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 886–898, 2019. https://doi.org/10.1145/3352460.3358317.