Paper Review Form (1000 words maximum)
cs940: Fast Byte-Granularity Software Fault Isolation [1]

March 5, 2018

Paper Summary

The Byte-Granularity Isolation (BGI) compiler was designed to provide low-overhead fault isolation for Windows kernel extensions, which operate in trusted kernel space but can cause serious system faults on malfunction. BGI resolved severe performance penalties in prior kernel extension isolation solutions by implementing fine-grained runtime byte-level verification of virtual memory accesses. Code insertion through a compiler extension, as well as a runtime interposition library were used to provide full isolation coverage within BGI's protection scope. Evaluations of BGI demonstrated its high efficiency, ability to prevent some faults outside the protection scope, as well as its practical bug finding capabilities.

Pros and Cons

I believe that the paper has the following positive features:

- The use of various extension code examples enabled clear and effective illustration of BGI's interactions between the compiler, the interposition library, and the target extension code.
- Low-level optimisations of table accesses utilised peephole optimisation and data parallelism commonly seen in general purpose compiler designs, demonstrating the wide applicability of these ideas.
- In addition to providing runtime fault isolation as its primary purpose, BGI's bug finding capabilities through static code analysis demonstrated significant potential as an engineering utility for kernel extension developers.

I believe that the paper has the following negative features:

- Kernel extensions ultimately interact very frequently with the kernel, as noted by the authors. Despite having vastly improved upon prior researches, BGI's 6.4% to 16% overhead can still be deemed unacceptable for kernel extensions driving performance-critical devices. Although its bug finding capabilities can help the developer to eliminate some bugs instead.

- As a tool focused on practicality, BGI's reduced performance when transforming other compilers' binaries implied that it may be necessary for kernel extension developers to replace the original compiler in their toolchain, which may be difficult to achieve for highly customised toolchains.
- A wide range of assumptions (e.g. number of untrusted domains supported) and optimisations (e.g. x64-specific byte handling changes) were specific to Windows or architecture, limiting BGI's adaptability to other platforms.

The Problem/Motivation

Modern operating systems interface with a wide range of devices, most of which require kernel extensions running under considerable privileges and accesses, as it would be impractical performance-wise to enforce user-mode checks on these modules. However, due to quality variances in these extensions supplied by device manufacturers, they have become a constant source of poor system reliability. Existing solutions such as fault containment [2], user-mode drivers [3], code annotations [4], or even rewriting kernels in type-safe languages [5] suffered from severe kernel overheads or involved significant re-implementation effort, and were thus deemed impractical by the authors. The authors instead sought to leverage architecture and platform features of Windows on x86/x64 to implement an optimised low-level fault isolation system, and excluded malicious extension code [6] from the threat model.

The Solution/Approach

BGI includes two components: a compiler extension to automatically insert runtime virtual memory access verification into the code, and an interposition library to provide runtime access verification capabilities. Access rights subject to verification include read, write, function call pointer accesses (for control flow integrity) and kernel object typing accesses (for dynamic type safety and interface use correctness). Kernel extensions are organised into either trusted (exempt from BGI control) or untrusted domains, with the ability to include multiple associated extensions within one domain. Access rights of untrusted kernel extensions are governed through access control lists (ACLs) assigned to 8-byte blocks of virtual memory, with memory footprint reduced through optimisations focused on compact storage and architecture-leveraged reductions in access. Runtime checks against ACLs are carried out by wrappers in the interposition library, with check failures triggering extension suspension to protect the wider system. A prototype "turn-it-off-and-on" fault recovery mechanism is also provided. BGI does not prevent malicious kernel extensions from reading sensitive kernel memory, as checking reads would have generated excessive overhead. The lack of synchronisation in fast path access right checks due to performance considerations also makes BGI x86/x64 to attacks exploiting race conditions.

Evaluation

The authors evaluated three aspects of BGI's capabilities: providing strong kernel extension isolation assurances; generating acceptable performance overheads; and finding bugs in real kernel extension code. In evaluating isolation assurances, the authors injected commonly found bugs into two common kernel extensions (FAT external file system driver and Intel PRO ethernet driver), achieving 98%-100% isolation of external "blue screen" faults, as well as 47%-60% isolation of hang kernel faults, which BGI was not designed to isolate, but managed to regardless due to associated runtime symptoms. Just under half of simulated faults were successfully recovered through the prototype recovery mechanism. During performance evaluations, PostMark benchmarking of CPU and I/O performance observed 10%-12% overhead in BGI, significantly outperforming the predecessor Nooks [7]. BGI's networking tests also showed a comparable or better performance than the predecessor SafeDrive [8]. Finally, during integration testing, BGI identified 28 bugs in widely used kernel extensions later fixed under responsible disclosure, demonstrating its practical effectiveness in this aspect. Evaluations conducted were comprehensive and practical.

Your Opinion

By avoiding additional work for kernel extension developers (beyond integrating a new compiler into their toolchains), BGI was implemented as a highly-usable replacement solution for protecting reliability-critical systems interfacing third party devices. However, in achieving a low overhead, its design decisions sacrificed the solution's compatibility with other platforms. From Microsoft's point of view, this was well justified in 2009. However, this idea also had great re-engineering potential for other kernels and hardware architectures.

Race conditions in runtime access right verification were later proved to be more likely than the authors anticipated, as demonstrated in a serious iOS race condition-triggered use-after-free bug [9] in 2016, enabling an iOS jailbreak.

Questions for the Authors

- While a lot of features and optimisations of BGI were specific to Windows and x64, with Window's extension to ARM platforms, has BGI been adapted for kernel extension fault isolation on ARM architectures?
- Due to the emergence of race condition exploits for circumventing memory access verifications, has there been a rethink in the approach to synchronisation in BGI operations with serious performance impact?

References

[1] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 45–58.

[2] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 75–88.

[3] A. Forin, D. Golub, and B. N. Bershad, *An I/O system for Mach 3.0*. Carnegie-Mellon University. Department of Computer Science, 1991.

[4] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, "The design and implementation of microdrivers," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1. ACM, 2008, pp. 168–178.

[5] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 128–139.

[6] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5. ACM, 1994, pp. 203–216.

[7] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 207–222.

[8] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, "Safedrive: Safe and recoverable extensions using language-based techniques," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 45–60.

[9] I. Beer, "XNU kernel UaF due to lack of locking in set_dp_control_port," October 2016. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=965