Principles and Implementation Techniques of Software-Based Fault Isolation

Principles and Implementation Techniques of Software-Based Fault Isolation

Gang Tan
The Pennsylvania State University
gtan@cse.psu.edu

the essence of knowledge
Boston — Delft

Foundations and Trends® in Privacy and Secruity

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

G. Tan. *Principles and Implementation Techniques of Software-Based Fault Isolation*. Foundations and Trends® in Privacy and Secruity, vol. 1, no. 3, pp. 137–198, 2017.

ISBN: 978-1-68083-345-4

Foundations and Trends® in Privacy and Secruity

Volume 1, Issue 3, 2017

Editorial Board

Editorial Scope

Topics

Foundations and Trends® in Privacy and Security publishes survey and tutorial articles in the following topics:

- Access control
- Accountability
- Anonymity
- Application security
- Artifical intelligence methods in security and privacy
- Authentication
- Big data analytics and privacy
- Cloud security
- Cyber-physical systems security and privacy
- Distributed systems security and privacy
- Embedded systems security and privacy
- Forensics
- Hardware security
- Human factors in security and privacy
- Information flow
- Intrusion detection
- Malware
- Metrics
- Mobile security and privacy
- Language-based security and privacy
- Network security
- Privacy-preserving systems
- Protocol security
- Security and privacy policies
- Security architectures
- System security
- Web security and privacy

Information for Librarians

Foundations and Trends® in Privacy and Security, 2017, Volume 1, 4 issues. ISSN paper version 2474-1558. ISSN online version 2474-1566. Also available as a combined paper and online subscription.

Contents

Principles and Implementation Techniques of Software-Based Fault Isolation

Gang Tan[1]

[1] *Pennsylvania State University; gtan@cse.psu.edu*

ABSTRACT

When protecting a computer system, it is often necessary to isolate an untrusted component into a separate protection domain and provide only controlled interaction between the domain and the rest of the system. Software-based Fault Isolation (SFI) establishes a logical protection domain by inserting dynamic checks before memory and control-transfer instructions. Compared to other isolation mechanisms, it enjoys the benefits of high efficiency (with less than 5% performance overhead), being readily applicable to legacy native code, and not relying on special hardware or OS support. SFI has been successfully applied in many applications, including isolating OS kernel extensions, isolating plug-ins in browsers, and isolating native libraries in the Java Virtual Machine. In this survey article, we will discuss the SFI policy, its main implementation and optimization techniques, as well as an SFI formalization on an idealized assembly language.

Gang Tan (2017), "Principles and Implementation Techniques of Software-Based Fault Isolation", Foundations and Trends® in Privacy and Secruity: Vol. 1, No. 3, pp 137–198. DOI: 10.1561/3300000013.

1

Introduction

One fundamental idea in protecting a computer system is to have multiple *protection domains* in the system (Lampson, 1974). Each domain has its own capabilities, according to the domain's trustworthiness. Since the introduction of protection rings and virtual memory in Multics (Schroeder and Saltzer, 1972; Saltzer, 1974), all modern operating systems are structured to have an OS protection domain, also known as the kernel mode, and multiple user-application domains, which are processes in the user mode; the OS domain can execute privileged instructions, set up virtual memory protection, and perform access control on resources; a user-application domain has to go through the OS domain via the system-call interface to perform privileged operations. Domains can communicate by message passing or via shared objects. The boundaries between protection domains ensure that errors in one domain do not affect other domains.

It is natural to use protection domains to isolate untrusted components of a system. For instance, a web browser should isolate browser plug-ins so that their malfunctioning would not lead to the crash or leakage of sensitive information of the browser. In the same vein, an operating system should isolate device drivers, which are often developed by third-party vendors and have a higher bug rate than the OS kernel. In many such situations, it is highly desirable to isolate untrusted components in separate protection domains, grant them a minimum set of privileges, and allow only controlled interaction between them and privileged protection domains (Provos *et al.*, 2003; Brumley and Song, 2004).

Many mechanisms are possible for implementing protection domains. Table 1.1 provides a comparison among common mechanisms that can provide application-level protection domains. Hardware-based virtualization puts components into separate *virtual machines* and relies on virtual machine boundaries for fault toleration and resource control. Process-based separation puts components into separate *OS processes* and relies on hardware-backed virtual memory for isolating processes. In both hardware-based virtualization and process-based separation, user-level instructions run unmodified at native speed and they are fully transparent in that no special compiler is needed to recompile applications, nor do they require developers to port their code. However, their downside is the high-performance overhead for context switching between domains. For instance, a process context switch may require the flushing of the Translation Lookaside Buffer (TLB), which is the cache for the translation from virtual to physical addresses; it also brings adverse effect to data and instruction caches. A virtual machine context switch is even more costly as it involves the switching between two OSes. Therefore, when components are tightly coupled and require frequent domain crossings, separating them via virtual machines or processes is often not adopted due to the high cost of context switches.

Another approach is through *language-based isolation*, which relies on safe high-level languages for isolation. This approach fine-grained, portable, and flexible. The Java Virtual Machine (JVM) and the Common Language Runtime (CLR, *Common Language Infrastructure (CLI)* 2006) enforce type-based isolation through a combination of static and dynamic checks. Languages such as E (Miller, 2006) and Joe-E (Mettler *et al.*, 2010; Krishnamurthy *et al.*, 2010) enforce a stronger level of isolation than Java through an object-capability model. Their downside is an overall loss of performance caused by dynamic checks. Techniques

Table 1.1: Comparison of isolation mechanisms.

	Context-switch overhead	Per-instruction overhead	Compiler support	Software engineering effort
Virtual machines	very high	none	no	none
OS processes	high	none	no	none
Language-based isolation	low	medium (dynamic) or none (static)	yes	high
SFI	low	low	maybe	none or medium

using pure static types (e.g., Morrisett *et al.*, 1999) have no runtime overhead, but require nontrivial support from developers and compilers. One significant downside of language-based isolation is that a single language model has to be adopted, meaning that the software-engineering effort to rewrite legacy C/C++ code is significant.

Software-based Fault Isolation (SFI) is a software-instrumentation technique at the machine-code level for establishing logical protection domains within a process. The main idea is to designate a memory region for an untrusted component and instrument dangerous instructions in the component to constrain its memory access and control transfer behavior; it is sometimes referred to as code *sandboxing*. In SFI, protection domains stay within the same process, incurring low overhead when switching between domains. As a result, it is especially attractive in situations when domain crossings are frequent (e.g., the interaction between a browser and a plug-in, or the interaction between an OS and a device driver). As we will discuss, SFI can be implemented in many ways: in a machine-code interpreter, in a machine-code rewriter, or inside a compiler. When SFI is implemented in a machine-code interpreter or rewriter, applications can run without porting by developers. In contrast, some porting effort may be required when SFI is implemented inside a compiler, as is the case with NaCl (Yee *et al.*, 2009).

First proposed by Wahbe *et al.* (1993), SFI has enjoyed many successes thanks to its runtime efficiency, strong guarantee, and ease of implementation. It has been implemented in several architectures, including MIPS (Wahbe *et al.*, 1993), x86-32 (Small, 1997; McCamant and Morrisett, 2006; Ford and Cox, 2008; Yee *et al.*, 2009; Zeng *et al.*, 2011; Zeng *et al.*, 2013), x86-64 (Sehr *et al.*, 2010; Deng *et al.*, 2015), and ARM (Sehr *et al.*, 2010; Zhao *et al.*, 2011; Zhou *et al.*, 2014). It has also been used in many applications, including isolating OS kernel modules (Small, 1997; Erlingsson *et al.*, 2006; Mao *et al.*, 2011; Castro *et al.*, 2009), isolating plug-ins in the Chrome browser (Yee *et al.*, 2009; Sehr *et al.*, 2010), and isolating native libraries in the Java Virtual Machine (Siefers *et al.*, 2010; Sun and Tan, 2012).

In this survey article on SFI, we will focus on the principles and common implementation techniques behind many SFI systems. Chapter 2 will give a concise definition of the SFI policy. The bulk of the article will be in chapter 3, which presents the implementation and optimization techniques that enforce the SFI policy. In chapter 4, we will formalize the main constraints enforced by SFI, through a formalization of an SFI verifier; a correctness proof of the verifier will also be discussed. We will briefly discuss future research directions in chapter 5 and cover stronger policies than fault isolation in chapter 6.

References

Abadi, M., M. Budiu, Ú. Erlingsson, and J. Ligatti. 2005. “Control-flow integrity”. In: *12th ACM Conference on Computer and Communications Security (CCS)*. 340–353.

Akritidis, P., C. Cadar, C. Raiciu, M. Costa, and M. Castro. 2008. “Preventing Memory Error Exploits with WIT”. In: *IEEE Symposium on Security and Privacy (S&P)*. 263–277.

Ansel, J., P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. Schuff, D. Sehr, C. Biffle, and B. Yee. 2011. “Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code”. In: *ACM Conference on Programming Language Design and Implementation (PLDI)*. 355–366.

Barth, A., C. Jackson, C. Reis, and G. Chrome. 2008. “The security architecture of the Chromium browser”. *Tech. rep.*

Bittau, A., P. Marchenko, M. Handley, and B. Karp. 2008. “Wedge: splitting applications into reduced-privilege compartments”. In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. 309–322.

Bruening, D., Q. Zhao, and S. Amarasinghe. 2012. “Transparent Dynamic Instrumentation”. In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. 133–144.

Brumley, D. and D. Song. 2004. "Privtrans: Automatically Partitioning Programs for Privilege Separation". In: *13th Usenix Security Symposium*. 57–72.

Candea, G., S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. 2004. "Microreboot — A Technique for Cheap Recovery". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 31–44.

Castro, M., M. Costa, and T. Harris. 2006. "Securing Software by Enforcing Data-flow Integrity". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–160.

Castro, M., M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. 2009. "Fast byte-granularity software fault isolation". In: *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 45–58.

Chen, S., J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. 2005. "Non-control-data attacks are realistic threats". In: *14th Usenix Security Symposium*. 177–192.

Common Language Infrastructure (CLI). 2006. 4th. Standard ECMA-335. Ecma International.

Criswell, J., A. Lenharth, D. Dhurjati, and V. Adve. 2007. "Secure virtual architecture: a safe execution environment for commodity operating systems". *SIGOPS Oper. Syst. Rev.* 41(6): 351–366.

Deng, L., Q. Zeng, and Y. Liu. 2015. "ISboxing: An Instruction Substitution Based Data Sandboxing for x86 Untrusted Libraries". In: *30th Inernational Conference on ICT Systems Security and Privacy Protection*. 386–400.

Dhurjati, D. and V. S. Adve. 2006. "Backwards-compatible array bounds checking for C with very low overhead". In: *International Conference on Software engineering (ICSE)*. 162–171.

Donovan, A., R. Muth, B. Chen, and D. Sehr. 2010. "PNaCl: Portable Native Client Executables (white paper)". http://src.chromium.org/viewvc/native_client/data/site/pnacl.pdf.

Erlingsson, Ú., M. Abadi, M. Vrable, M. Budiu, and G. Necula. 2006. "XFI: Software Guards for System Address Spaces". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 75–88.

Ford, B. and R. Cox. 2008. "Vx32: Lightweight User-level Sandboxing on the x86". In: *USENIX Annual Technical Conference*. 293–306.

Furr, M. and J. Foster. 2005. "Checking type safety of foreign function calls." In: *ACM Conference on Programming Language Design and Implementation (PLDI)*. 62–72.

"Intel Software Guard Extensions (Intel SGX)". 2016. https://software.intel.com/en-us/sgx.

Jaleel, A. 2010. "Memory characterization of workloads using instrumentation-driven simulation". URL: http://www.jaleels.org/ajaleel/workload/SPECanalysis.pdf.

Kiriansky, V., D. Bruening, and S. Amarasinghe. 2002. "Secure Execution via Program Shepherding". In: *11th Usenix Security Symposium*. 191–206.

Kondoh, G. and T. Onodera. 2008. "Finding bugs in Java Native Interface programs". In: *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM. 109–118.

Krishnamurthy, A., A. Mettler, and D. Wagner. 2010. "Fine-grained privilege separation for web applications". In: *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. 551–560.

Kroll, J. A., G. Stewart, and A. W. Appel. 2014. "Portable Software Fault Isolation". In: *CSF*. 18–32.

Kuznetsov, V., L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. 2014. "Code-Pointer Integrity". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–163.

Lampson, B. W. 1974. "Protection". *SIGOPS Oper. Syst. Rev.* 8(1): 18–24.

Leroy, X. 2006. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant". In: *33rd ACM Symposium on Principles of Programming Languages (POPL)*. 42–54.

Liu, S., G. Tan, and T. Jaeger. 2017. "PtrSplit: Supporting General Pointers in Automatic Program Partitioning". In: *24th ACM Conference on Computer and Communications Security (CCS)*. To appear.

Mao, Y., H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. 2011. "Software fault isolation with API integrity and multi-principal modules". In: *SOSP*. 115–128.

McCamant, S. 2006. "A machine-checked safety-proof for a CISC-compatible SFI technique". *Tech. rep.* No. 2006-035. MIT Computer Science and Artificial Intelligence Laboratory.

McCamant, S. and G. Morrisett. 2006. "Evaluating SFI for a CISC Architecture". In: *15th Usenix Security Symposium*.

Mettler, A., D. Wagner, and T. Close. 2010. "Joe-E: A Security-Oriented Subset of Java". In: *Network and Distributed System Security Symposium (NDSS)*.

Miller, M. 2006. "Robust composition: towards a unified approach to access control and concurrency control". *PhD thesis*. Baltimore, MD: Johns Hopkins University.

Morrisett, G., G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. 2012. "RockSalt: Better, Faster, Stronger SFI for the x86". In: *ACM Conference on Programming Language Design and Implementation (PLDI)*. 395–404.

Morrisett, G., D. Walker, K. Crary, and N. Glew. 1999. "From System F to Typed Assembly Language". *ACM Transactions on Programming Languages and Systems*. 21(3): 527–568.

Nagarakatte, S., J. Zhao, M. M. K. Martin, and S. Zdancewic. 2009. "SoftBound: highly compatible and complete spatial memory safety for C". In: *ACM Conference on Programming Language Design and Implementation (PLDI)*. 245–258.

"Native Client Security Contest". 2009. https://developer.chrome.com/native-client/community/security-contest.

Necula, G., S. McPeak, and W. Weimer. 2002. "CCured: type-safe retrofitting of legacy code". In: *29th ACM Symposium on Principles of Programming Languages (POPL)*. Portland, Oregon. 128–139.

Niu, B. and G. Tan. 2013. "Monitor Integrity Protection with Space Efficiency and Separate Compilation". In: *20th ACM Conference on Computer and Communications Security (CCS)*.

Payer, M. and T. Gross. 2011. “Fine-grained user-space security through virtualization”. In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual execution Environments (VEE ’11)*. 157–168.

Provos, N., M. Friedl, and P. Honeyman. 2003. “Preventing privilege escalation”. In: *12th Usenix Security Symposium*. 231–242.

Saltzer, J. H. 1974. “Protection and the Control of Information Sharing in Multics”. *Communications of the ACM*. 17(7): 388–402.

Schroeder, M. D. and J. H. Saltzer. 1972. “A Hardware Architecture for Implementing Protection Rings”. *Communications of the ACM*. 15(3): 157–170.

Scott, K. and J. Davidson. 2002. “Safe Virtual Execution Using Software Dynamic Translation”. In: *Proceedings of the 18th Annual Computer Security Applications Conference. ACSAC ’02*. 209–218.

Sehr, D., R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. 2010. “Adapting Software Fault Isolation to Contemporary CPU Architectures”. In: *19th Usenix Security Symposium*. 1–12.

Seltzer, M. I., Y. Endo, C. Small, and K. A. Smith. 1996. “Dealing with Disaster: Surviving Misbehaved Kernel Extensions”. In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 213–227.

Shacham, H. 2007. “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)”. In: *14th ACM Conference on Computer and Communications Security (CCS)*. 552–561.

Siefers, J., G. Tan, and G. Morrisett. 2010. “Robusta: Taming the Native Beast of the JVM”. In: *17th ACM Conference on Computer and Communications Security (CCS)*. 201–211.

Small, C. 1997. “A tool for constructing safe extensible C++ systems”. In: *COOTS’97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*. 174–184.

Sun, M. and G. Tan. 2012. “JVM-Portable Sandboxing of Java’s Native Libraries”. In: *17th European Symposium on Research in Computer Security (ESORICS)*. 842–858.

Tan, G. and J. Croft. 2008. “An empirical security study of the native code in the JDK”. In: *17th Usenix Security Symposium*. 365–377.

Wahbe, R., S. Lucco, T. Anderson, and S. Graham. 1993. “Efficient Software-Based Fault Isolation”. In: *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. New York: ACM Press. 203–216.

Yee, B., D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. 2009. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. In: *IEEE Symposium on Security and Privacy (S&P)*.

Yongzheng Wu Jun Sun, Y. L. and J. S. Dong. 2013. “Automatically partition software into least privilege components using dynamic data dependency analysis”. In: *International Conference on Automated Software Engineering (ASE)*. 323–333.

Zeng, B., G. Tan, and Ú. Erlingsson. 2013. “Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors”. In: *22nd Usenix Security Symposium*. 369–382.

Zeng, B., G. Tan, and G. Morrisett. 2011. “Combining control-flow integrity and static analysis for efficient and validated data sandboxing”. In: *18th ACM Conference on Computer and Communications Security (CCS)*. 29–40.

Zhang, Y., A. Juels, M. K. Reiter, and T. Ristenpart. 2012. “Cross-VM Side Channels and Their Use to Extract Private Keys”. In: *19th ACM Conference on Computer and Communications Security (CCS)*. 305–316.

Zhao, L., G. Li, B. D. Sutter, and J. Regehr. 2011. “ARMor: Fully Verified Software Fault Isolation”. In: *11th Intl. Conf. on Embedded Software*. ACM.

Zhou, Y., X. Wang, Y. Chen, and Z. Wang. 2014. “ARMlock: Hardware-based Fault Isolation for ARM”. In: *21st ACM Conference on Computer and Communications Security (CCS)*. 558–569.