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

Full text available at: http://dx.doi.org/10.1561/3300000013
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
- Artificial 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

1 Introduction 2

2 The SFI Policy 6

3 SFI Enforcement 9
   3.1 Enforcing the data-access policy ......................... 13
      3.1.1 Data region specialization .......................... 14
      3.1.2 Integrity-only isolation ............................. 15
      3.1.3 Address masking ..................................... 16
      3.1.4 Guard zones .......................................... 18
      3.1.5 Guarding changes instead of uses .................... 22
      3.1.6 Finding scratch registers ........................... 23
      3.1.7 Architecture-specific optimizations .................. 24
      3.1.8 Applicability in dynamic binary translation .......... 24
   3.2 Enforcing the control-flow policy ......................... 25
      3.2.1 Indirect-jump control-flow enforcement ............... 27
      3.2.2 Interaction with the outside world .................. 31
   3.3 Portable enforcement ..................................... 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>SFI Verification and Formalization</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>Operational semantics</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Modeling an SFI verifier</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Verifier correctness</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>Future Directions</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>Going Beyond Fault Isolation</td>
<td>52</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>55</td>
</tr>
<tr>
<td>8</td>
<td>Acknowledgments</td>
<td>56</td>
</tr>
</tbody>
</table>
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.
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 devel-
oped 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.

<table>
<thead>
<tr>
<th></th>
<th>Context-switch overhead</th>
<th>Per-instruction overhead</th>
<th>Compiler support</th>
<th>Software engineering effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual machines</td>
<td>very high</td>
<td>none</td>
<td>no</td>
<td>none</td>
</tr>
<tr>
<td>OS processes</td>
<td>high</td>
<td>none</td>
<td>no</td>
<td>none</td>
</tr>
<tr>
<td>Language-based isolation</td>
<td>low</td>
<td>medium (dynamic) or none (static)</td>
<td>yes</td>
<td>high</td>
</tr>
<tr>
<td>SFI</td>
<td>low</td>
<td>low</td>
<td>maybe</td>
<td>none or medium</td>
</tr>
</tbody>
</table>

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.


