# 50 Years of Isolation

Goonasekera N, Caelli W.J and Sahama T
*Faculty of Science and Technology*
*Queensland University of Technology*
*Brisbane, Australia*
nuwan.goonasekera@student.qut.edu.au, w.caelli@qut.edu.au, t.sahama@qut.edu.au

## Abstract

*The traditional means for isolating applications from each other is via the use of operating system provided "process" abstraction facilities. However, as applications now consist of multiple fine-grained, independent and separately acquired components, the traditional process abstraction model is proving to be insufficient in ensuring this isolation. Statistics indicate that a high percentage of software failure occurs due to propagation of component failures. These observations are further bolstered by the attempts by modern Internet browser application developers, for example, to adopt multi-process architectures in order to increase robustness. Therefore, a fresh look at the available options for isolating program components is necessary and this paper provides an overview of previous activity and current research in the area.*

## 1. Introduction

Many applications extend their functionality by loading components dynamically into their allocated address space. For example, operating system kernels may dynamically load device drivers, web browsers may load "extensions" and many applications support some form of "plug-in" components, often of unknown origin and quality, to augment their functionality. However, such functionality may come at a cost whereby the failure of a single component can adversely affect the entire host application.

In contrast to software components, processes are typically protected from each other so that buggy or malicious code may not readily affect an entire system. Traditionally, operating systems have provided support for process isolation through the following means, as mentioned by Law and McCann [1].

a. Preventing applications from executing privileged instructions (e.g.. disabling interrupts, executing direct I/O instructions, etc.) and
b. Preventing an application from accessing illegal memory locations (e.g. another application's memory, memory mapped I/O locations, etc.)

Most operating systems today enable this isolation through use of processor modes and memory paging [2], while earlier computer designs such as memory segmentation and typing structures have been largely ignored. However, the more recent notion of an application-oriented, software component does not have the ability to take advantage of similar isolation schemes. Typically, an application will load a component into its existing address space, enabling the component to access any part of the host application's memory, thus leaving the host vulnerable to bad or misbehaving applications. Small and Seltzer [3] argue that if a component consistently crashes its host, the extra functionality is hardly worthwhile. Despite this, they note that few component extensions address the reliability issue.

The statistics are quite revealing: over 85% of Windows XP crashes are due to faulty device drivers and Linux drivers may have 2 to 7 times the bug count of the kernel [4]. Such failures are not limited to OS kernel drivers and applications suffer similar problems. Zeigler [5] indicates that over 70% of Internet Explorer crashes are due to 3rd party add-ons. To combat this issue, both Internet Explorer [5] and Google's Chrome browser [6] have shifted to multi-process architectures where program components are isolated into several, disparate Operating System (OS) processes.

Xu et al [7] note that dynamically loaded extensions need to be verified as the hosting process needs to be assured that an untrusted component is safe and does not compromise the host's or other components' integrity. Clearly, a revisitation of component isolation

978-0-7695-3737-5/09 $25.00 © 2009 IEEE
DOI 10.1109/UIC-ATC.2009.86

mechanisms and a holistic view to combating the problem are pressing requirements. This paper provides a review of various attempts at software component isolation, past and present, with insights into suitable mechanisms for such enforceable and reliable isolation.

## 2. A categorization of isolation mechanisms

This section provides an overview of more current mechanisms available for component isolation. Three basic mechanisms for isolation are identified and categorized by Small and Seltzer [3]. We expand this to a more fine-grained categorization, to reflect the variety of isolation mechanisms available, set out in relation to their level within the computer's architecture. This categorization will be the basis for discussing those various isolation mechanisms. It should be noted however, that some mechanisms do not fit cleanly into a category and may be a mix of many techniques.

| Isolation Mechanism | Description | Examples |
|---|---|---|
| Hardware Isolation | This category deals with mechanisms that utilize some hardware support in enforcing isolation. The main drawback is that, due to reliance on specific hardware features, these techniques may not be portable across all computer architectures | Privilege level change, Isolation using memory segmentation and/or typing support, Isolation using paging hardware, Isolation into separate processors, e.g. peripheral I/O processors, Hardware virtualization support. |
| Binary code level isolation | Protection afforded by modifying binary code. | Software Fault Isolation, Binary translation, Virtual Machine Monitors. |
| Integration into OS kernel isolation facilities | OS kernel protection mechanisms for isolating components. | Kernel Wrapping, Sandboxing. |
| Language support | Isolation provided through language level/compiler level support. | Type safe languages, Static analysis, Compilers. |
| Application level isolation | Isolation facilities implemented entirely in user space. | Interpreters, JIT Compilers, Vx32, Native Client. |

**Table 1.** Categorization of isolation mechanisms and examples

### 2.1 Hardware Isolation

A variety of hardware based techniques have been utilized for process/component isolation for almost 50 years, dating from the earliest 2nd generation computer systems providing multiprogramming facilities, e.g. English Electric KDF-9 etc. Some techniques, such as simple, dual-state privilege level change based on OS-user separation, are available in most modern system architectures. Some other techniques, such as memory segmentation support, are now only widely available on certain architectures like the Intel "*x86*" line of processors. This section examines each of these mechanisms in turn, highlighting their usage and specific advantages and disadvantages.

The idea of protection rings and segmentation were pioneered in the Multics system and implemented in the GE-645 computer [8]. Earlier 2nd and 3rd generation mainframe computer architectures did, however, provide separate hardware mechanisms to assist in a form of isolation. These varied greatly from complete isolation into distinct and separate processor units, as in the peripheral processor concepts in the Control Data 6000, 7000, Cyber-70/170 computer systems and memory "tagging" used in the IBM System/360 series. Amdahl [9] et al clearly pointed out in 1964 that program isolation was essential in the System/360 in terms of "*tamper-proof storage protection*" and a "*protected supervisor program*". The Burroughs B5000, introduced in 1961 was also an early system which featured segmentation and tagged memory [10].

Most modern operating systems utilize a 2 level protection mechanism [2]. The operating system executes at a higher privilege level (ring) allowing it to execute any instruction. Applications have a lower privilege level and hardware ensures that any attempt to execute a high privilege instruction causes a "trap"[2]. The question of whether or not such a 2-level scheme is sufficient for complete isolation of processes to provide trusted system operation is one of constant debate.

A more specific version of privilege level based protection is utilized in micro-kernel based operating systems [2]. This minimalist approach is an attempt to enforce the principle of least privilege and economy of mechanism [11]. At its extremes, even paging and scheduling may run as user mode applications [12],

which allows for a more modular approach with a greater degree of robustness and isolation of faults.

The Minix 3 operating system provides a recent example of this approach [12]. User space components are monitored by a "*reincarnation server*", a process which periodically polls each component to see whether it is healthy [13]. If the component is found to be defective, it is "*reincarnated*", by sending it a *kill* signal and restarting the process shortly thereafter [13].

The major drawback of the micro-kernel approach is the high cost of switching protection domains. Estimates for switching overhead over traditional monolithic kernels run as high as two orders of magnitude [14] though proponents of the micro-kernel approach have demonstrated to the contrary that proper optimization can lead to very low overheads [15].

Overall, the level of protection afforded by this mechanism is more suited to process level granularity. For components, which may be at much finer levels of granularity, the protection domain switching overhead may be too prohibitive. This is sharply highlighted by the fact that many traditional monolithic kernels still continue to minimize privilege level changes specifically because of such perceived overheads [16].

Another widely used isolation mechanism is one of memory "paging". Paging protection is a feature offered by most modern computers and works by dividing the address space into pages, typically of fixed-length, with the ability to set permissions per page [17]. Most modern operating systems utilize paging protection for enabling process isolation [2].

Peterson et al [18] analyse the options available to designers of sandboxing mechanisms and describe a generic operating system Application Programming Interface (API) for creating sandboxed programs, where each sandboxed process runs in a separate address space by utilizing paging hardware. However, the main drawback in using paging support for fine-grained components is that pages may be too coarsely grained for component protection [19], typically being in the order of a few Kilobytes (KiB), with 4KiB being the most common. Components however, typically occupy only a few hundred bytes [20].

Memory "segmentation" hardware support is another important protection mechanism used over the last 50 years as highlighted by use in the Multics system. In the Intel x86 hardware architecture, each segment can have 4 possible segment privilege levels (SPL). The hardware ensures that lower privilege segments cannot access higher privilege segments, thus isolating memory from each other. The advantages of such hardware support for preserving high performance are stressed by Chiueh et al [21].

Banerji et al [22] utilize kernel/user mode separation along with paging and segmentation hardware to create Protected Shared Libraries (PSLs). Additional examples of using segments for component level isolation are described later.

However, a significant problem in using segmentation is the gradual dwindling of support for this hardware feature even in Intel processor units [23].

Hardware virtualization is also an isolation mechanism which has been around for decades. The concept of virtual machines go back into the 1950s/60s, e.g. in early computer systems from the United Kingdom, and such facilities were implemented in the likes of the IBM System/360 Model 67 and System/370 series [24]. System/370 featured hardware support for interpretive execution, making the development of VMM software much simpler [24]. Despite this however, the popularity of VM technology waned somewhat over the years, but it has lately gained a resurgence of interest with the development and marketing of software systems such as VMware [25], which provided a VMM for the popular Intel x86 architecture, despite the fact that the Intel x86 architecture itself had several non-virtualizable instructions [26]. Many novel techniques have been used to overcome these limitations, such as binary translation [25] and para-virtualization [27].

In 2005, Intel and AMD introduced additional machine instructions to their respective architectures to remedy this problem [23]. The Intel and AMD extensions are similar [25]. However, as noted by Adams and Agesen [25], early versions of Intel's and AMD's hardware virtualization did not necessarily result in better performance, due to the lack of support for Memory Management Unit (MMU) virtualization. To remedy this, AMD introduced Nested Page Tables (NPT) [28] and Intel has followed suit by adding support for Extended Page Tables (EPT) in their new "*Nehalem*" processor architecture, both of which add support for MMU virtualization [29].

## 2.2 Binary Code Level Isolation

Binary code level isolation relies on modifying the application binary at load-time or run-time, in order to insert additional checks and guards for ensuring isolation.

One of the key techniques in binary code level isolation is Software Fault Isolation (SFI). This method was first described by Wahbe et al [16] and the basic technique has been utilized in many forms. A "*sandboxed*" code version is created so that memory references always fall within the sandboxed region,

thus preventing a component from accessing memory outside of its bounds [16]. SFI, originally demonstrated by Wahbe et al on a Reduced Instruction Set Computer (RISC) architecture, has also been demonstrated on Complex Instruction Set Computer (CISC) architectures [30]. Techniques such as binary translation [25] are offshoots of the ideas in SFI.

The main advantages of this method, as identified by Xu et al [7] are

a. Operates directly on binary code,
b. Provides the ability to extend the host at a very fine-grained level,
c. Enforces a default collection of safety conditions to prevent array out-of-bounds violations, address-alignment violations, uses of uninitialized variables, null-pointer dereferences and stack manipulation violations

However, one significant weakness in their approach is that it requires compiler level modifications for the technique to work. As they point out, modification of the executable binary is complicated and adds too much overhead to the code injection process. However, Erlingsson et al [31] have attempted to address this problem by using control flow analysis and a binary rewriter which ensures that all expected properties and guards continue to hold.

SFI techniques have also been used in the "Nooks" system [4], in combination with hardware support, to create an architecture for device driver fault isolation and recovery. Fraser et al [32] describe similar protection mechanisms for "commercial-off-the-shelf" or COTS systems. Kumar et al [33] describe the use of SFI in embedded systems, where hardware support for protection domains is absent. In addition, Small and Seltzer [3] have estimated the performance characteristics of various techniques, and conclude that SFI based techniques offer good overall performance.

A strong example of such software based techniques providing better performance than corresponding hardware protection comes from Adams and Agesen [25]. Through their experience in implementing the popular VMware virtual machine monitor, they provide performance measurements which indicate that hardware assisted techniques can be overshadowed by Binary Translation techniques.

Swift et al [4] point out that it may be difficult to implement SFI when the range of addresses are not contiguous. Further, although it is relatively cheap to call into SFI code as opposed to a protection domain switch, the SFI code itself executes more slowly due to the additional checks.

## 2.3 Integration into OS Kernel Isolation Facilities

Attempts have been made to enforce isolation through integrating separate application level components with the OS kernel API layer. One such example is the use of "wrapping" techniques. Wrapping involves the verification of all parameters passed between the host and its extensions [4]. In the Nooks architecture [4], an amalgamation of techniques such as hardware memory protection, software fault isolation and privilege lowering along with kernel wrapping are used to prevent device driver failures. Each device driver is carefully wrapped by a proxy which is responsible for fault isolation and recovery [4].

However, Tanenbaum et al [12] points out that attempting to define and create a wrapper for use around each and every device driver is an error prone and painful process, hampering the adoption of the technique. Further, Erlingsson et al [31] point out that the protection offered by Nooks can be easily circumvented by malicious code.

Peterson et al [18] describe a generic operating system API for creating sandboxed programs, where each sandboxed process runs in separate address spaces. Their work lends support to the need for making components a first class concept within the operating system, as argued by Mendelsohn [34]. An interesting implementation is also made in the Go! OS [1]. Instead of weaving in and out of kernel/user mode, the Go! component based OS works entirely in kernel mode.

## 2.4 Language Support

Language based isolation relies on the safety of a language's type system, where the operations that a program performs can only be operations that are deemed sensible for that type [35]. Typically, this will involve both a dynamic and a static access control mechanism.

The SPIN operating system utilizes Modula-3 as a type-safe programming language along with a trusted compiler to create type-safe extensions [36]. A more modern example of the use of type safe code for component isolation is the "Singularity" operating system, a prototype OS created by Microsoft Research [37]. The key philosophy behind Singularity is the concept of a Software Isolated Process (SIP), which, unlike traditional hardware based process isolation, relies on static type checking and language safety rules to ensure isolation between processes. The results

indicate that SIP may incur lower overheads than hardware based isolation [37].

In Singularity, all software components have to be rewritten in a type safe language (in this case a .NET compatible one) in order for the scheme to work, making it unsuitable for the large base of existing applications [37]. In addition, Swift et al [4] point out several difficulties in the adoption of type-safe languages. The major issue is the problem of rewriting all drivers in the type safe language.

Another approach is that of "Proof Carrying Code (PCC)", whereby an automated proof generator is used to analyze each program and attach a formal proof that the program will execute within its defined boundaries [38]. However, writing a comprehensive proof generator which can deal with the complexities of optimized code remains a problem and so far, the technique has not been demonstrated with non-trivial examples [12]. Guaranteeing the completeness of the policy itself is also difficult [39] and therefore, this technique remains open to further investigation.

### 2.5 Application Level Isolation

Application Level Isolation involves isolation enforced entirely in user-space and managed by the application itself. Interpretation based isolation has been categorized under application level support, as it is usually performed entirely in user-space. Interpreted languages have shown excellent safety properties and can be made extremely secure [12]. For example, the JVM contains a built-in verifier that provides several safety checks to ensure that no forged pointers or pointer manipulations can be performed, effectively preventing code from accessing unauthorized memory locations [12, 40]. The major drawback of interpretation techniques is speed.

A further problem is that not all programs can be written in an interpreted language. For example, a badly written extension DLL can easily crash an entire "Java Virtual Machine (JVM)", as they all reside within the same address space. One solution to this issue, is to isolate components in separate process address spaces [41]. Lastly, there is a large existing base of code that is in binary format. It is not feasible to rewrite all of these programs in an interpreted language [12].

Another technique implemented at an application level is that of using a multi-process application architecture. This has been exemplified by the current trend of browsers being built using such a model [6, 42]. The basic idea is to use the operating system's IPC mechanisms to communicate between components loaded into disparate OS processes. A trade off is made between performance and reliability [6].

A somewhat similar attempt is an application level library for isolating components through a combination of SFI and segmentation hardware [43]. This attempt is novel in that the entire library is implemented in user-mode, requiring no changes to the OS kernel. It is limited by its reliance on x86 architectural features. A similar attempt is reported in the Google Native Client project, which enables the creation of browser extensions using portable x86 binary components [44].

## 3. Conclusion

This literature survey has evaluated the current landscape of component isolation mechanisms. Through this survey, we draw the following conclusions about the various protection mechanisms.

Privilege-level change offers a flexible mechanism for detecting illegal instructions, but MMU support is required for detecting illegal memory accesses. The cost of ring transitions remains very high however, and the cost is prohibitive for the extremely fine granularity of protection required for component level protection.

Paging hardware provides a flexible means of protecting illegal memory access but is somewhat coarsely grained to be really useful for component level protection.

Segmentation hardware appears very promising for component level protection facilities as recently demonstrated by its use in Vx32 and Google Native Client. However, not all machine architectures support segmentation, limiting the portability of the solution.

Hardware virtualization support offers great promise in implementing isolation schemes, especially with the introduction of the AMD-V Nested Paging and Intel Nehalem extensions. Although originally designed for the execution of full operating systems within a virtual machine, it may be possible to construct light-weight isolation domains for individual components by using these additional instructions.

Software Fault Isolation is a promising technique for ensuring component safety and protection that offers good overall performance. However, most current implementations require compiler level modifications and do not work directly on existing binary code, rendering its use difficult.

Static analysis is marred by the problem of being difficult to implement at a binary code level and due to the difficulty in ensuring that the technique is failsafe. Overall, static analysis could be deemed a preventive measure and mainly be used in determining whether a

given component obeys certain constraints during its execution. Thus, it could be used to prevent a faulty component from being loaded in the first place, but not to prevent failure during the execution of the component.

Proof-carrying code is yet to be demonstrated with non-trivial examples. However, it is a promising avenue for further research.

Type-safe code provides an excellent compromise between safety and performance. However, it cannot be applied to existing binary code and requires a complete rewrite in a type-safe language, rendering the technique difficult to apply to the massive code bases already written.

Wrapping is difficult to apply for the ensuring of complete protection as the current difficulty with wrapping is the painstaking process of manually writing a hardware straightjacket for each and every component.

Interpretation techniques are very effective for light-weight component isolation. However, the technique is difficult to apply at a binary code level. It is also not suitable for kernel level extensions, as the required timing granularity is too fine. Finally, it mandates the use of specific languages, which leaves out a large base of binary components.

Application Level Isolation such as that used in Google Chrome utilize the relatively tried and tested methods of IPC, but also incur the associated performance penalties of IPC mechanisms. There is also an increase in complexity due to the extra effort required in coordinating between components.

Finally, further research is needed into many of the above technologies if components are to become first-class concepts within operating systems. At the same time, the needs of autonomic software systems can only be met if underlying hardware and operating system/middleware component software systems can rely on those lower layer isolation schemes to provide the verifiable security functions needed.

## 4. References


[1] G. Law and J. McCann, "A new protection model for component-based operating systems," in *Proceedings of the IEEE International Performance, Computing, and Communications Conference*, 2000, pp. 537-543.

[2] A. S. Tanenbaum, *Modern Operating Systems*, 2nd ed.: Prentice Hall, 2001.

[3] C. Small and M. Seltzer, "A comparison of OS extension technologies," in *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, 1996, pp. 41-54.

[4] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, "Nooks: an architecture for reliable device drivers," in *Proceedings of the 10th ACM SIGOPS European workshop: beyond the PC* Saint-Emilion, France: ACM Press, 2002.

[5] A. Zeigler. (2009, Jan 30). IE8 and Loosely-Coupled IE (LCIE). [Online]. Available: http://blogs.msdn.com/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx

[6] A. Barth, C. Jackson, C. Reis, and The Google Chrome Team. (2009 Jan. 30). The Security Architecture of the Chromium Browser. Available: http://crypto.stanford.edu/websec/chromium/

[7] Z. Xu, B. P. Miller, and T. Reps, "Safety checking of machine code," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, Vancouver, BC, Canada, 2000, pp. 70-82.

[8] J. H. Saltzer, "Protection and the control of information sharing in Multics," *Commun. ACM,* vol. 17, pp. 388-402, 1974.

[9] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development,* vol. 8, 1964.

[10] A. J. W. Mayer, "The architecture of the Burroughs B5000: 20 years later and still ahead of the times?," *SIGARCH Computer Architecture News,* vol. 10, pp. 3-10, 1982.

[11] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE,* vol. 63, pp. 1278-1308, 1975.

[12] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?," *Computer,* vol. 39, pp. 44-51, 2006.

[13] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a Highly Dependable Operating System," in *Sixth European Dependable Computing Conference*, 2006, pp. 3-12.

[14] A. Purohit, C. P. Wright, J. Spadavecchia, and E. Zadok, "Cosy: Develop in User-Land, Run in Kernel-Mode," in *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems* Lihue, Hawaii, USA: USENIX Association, 2003.

[15] J. Liedtke, K. Elphinstone, S. Schonberg, H. Hartig, G. Heiser, N. Islam, and T. Jaeger, "Achieved IPC performance (still the foundation for extensibility)," in *The Sixth Workshop on Hot Topics in Operating Systems*, 1997, pp. 28-31.

[16] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *SIGOPS Operating Systems Review,* vol. 27, pp. 203-216, 1993.

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed.: Morgan Kaufmann Publishers Inc., 2006.

[18] D. S. Peterson, M. Bishop, and R. Pandey, "A Flexible Containment Mechanism for Executing Untrusted Code," in *Proceedings of the 11th USENIX Security Symposium*: USENIX Association, 2002.

[19] J. Shen, G. Venkataramani, and M. Prvulovic, "Tradeoffs in fine-grained heap memory protection," in *Proceedings of the 1st workshop on Architectural and*

*system support for improving software dependability* San Jose, California: ACM Press, 2006.

[20] L. Lam and T. Chiueh, "Checking array bound violation using segmentation hardware," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2005, pp. 388-397.

[21] T. Chiueh, G. Venkitachalam, and P. Pradhan, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," in *Proceedings of the seventeenth ACM symposium on Operating systems principles* Charleston, South Carolina, United States: ACM Press, 1999.

[22] A. Banerji, J. M. Tracey, and D. L. Cohn, "Protected shared libraries - a new approach to modularity and sharing," *Proceedings of the USENIX 1997 Annual Technical Conference,* pp. 59-75, 1997.

[23] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual* vol. 3B: System Programming Guide: Intel Corporation, 2007.

[24] P. H. Gum "System/370 Extended Architecture: Facilities for Virtual Machines," *IBM Journal of Research and Development,* vol. 27, 1983.

[25] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM SIGARCH Computer Architecture News,* vol. 34, pp. 2-13, 2006.

[26] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor," in *Proceedings of the 9th conference on USENIX Security Symposium*, Denver, Colorado, 2000, p. 10.

[27] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review,* vol. 37, pp. 164-177, 2003.

[28] AMD. (2009 Jan. 30). AMD-V™ Nested Paging. Advanced Micro Devices, Inc., [Online]. Available: http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf

[29] Intel. (2009, Jan. 30). Intel® Virtualization Technology. [Online]. Available: http://www.intel.com/technology/virtualization/index.htm

[30] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proceedings of the 15th conference on USENIX Security Symposium*. vol. 15 Vancouver, B.C., Canada: USENIX Association, 2006.

[31] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software Guards for System Address Spaces " *Symposium on Operating System Design and Implementation,* pp. 75–88 2006.

[32] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS software with generic software wrappers," in *Foundations of Intrusion Tolerant Systems [Organically Assured and Survivable Information Systems]*, 2003, pp. 399-413.

[33] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: Software-based Memory Protection For Sensor Nodes," in *6th International Symposium on Information Processing in Sensor Networks*, 2007, pp. 340-349.

[34] N. Mendelsohn, "Operating systems for component software environments," in *The Sixth Workshop on Hot Topics in Operating Systems*, 1997, pp. 49-54.

[35] C. Hawblitzel and T. von Eicken, "A Case for Language-Based Protection," Cornell University Technical Report 98-1670, 1998.

[36] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility safety and performance in the SPIN operating system," in *Proceedings of the fifteenth ACM symposium on Operating systems principles* Copper Mountain, Colorado, United States: ACM, 1995.

[37] M. Aiken, M. Fahndrich, C. Hawblitzel, G. Hunt, and J. Larus, "Deconstructing process isolation," in *Proceedings of the 2006 workshop on Memory system performance and correctness* San Jose, California: ACM Press, 2006.

[38] G. C. Necula and P. Lee, "Safe Kernel Extensions Without Run-Time Checking," in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, 1996.

[39] I. Ganev, G. Eisenhauer, and K. Schwan, "Kernel plugins: when a VM is too much," in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, San Jose, California, 2004.

[40] Y. Chiba, "Heap protection for Java virtual machines," in *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java* Mannheim, Germany: ACM Press, 2006.

[41] G. Czajkowski and L. Dayn, "Multitasking without compromise: a virtual machine evolution," in *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* Tampa Bay, FL, USA: ACM, 2001.

[42] C. Reis, B. Bershad, S. D. Gribble, and H. M. Levy, "Using Processes to Improve the Reliability of Browser-based Applications," Technical Report UW-CSE-2007-12-01, Department of Computer Science and Engineering, University of Washington, 2007.

[43] B. Ford and R. Cox, "Vx32: Lightweight User-level Sandboxing on the x86," in *USENIX Annual Technical Conference*, Boston, MA, 2008, pp. 293–306.

[44] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *To appear in the Proceedings of the IEEE Symposium on Security and Privacy*, 2009.