Separating Information Protection from Resource Management

by

Jisoo Yang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2010

Doctoral Committee:

Professor Kang G. Shin, Chair
Professor Peter M. Chen
Associate Professor Jason N. Flinn
Assistant Professor Mark W. Newman

To my mother

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Professor Kang G. Shin, for his support and guidance throughout the years at the University of Michigan. This dissertation would not have been possible without his consistent support and encouragement. I also would like to thank Professor Peter Chen, Professor Jason Flinn and Professor Mark Newman for serving on my dissertation committee, and their invaluable comments and feedbacks.

I am grateful to all the members of Real-Time Computing Laboratory for their support and friendship. I especially would like to thank the members of the systems group, Songkuk Kim, Hai Huang, Howard Tsai, Pradeep Padala, Karen Hou, Thuy Vu and Xiaoen Ju for their suggestions and discussions. Special thanks should go to Min-Gyu Cho, Kyu-han Kim, Sangsoo Park, and Antino Kim, who were more than willing to help and support me, and made my life in Ann Arbor much more fun and enjoyable. I also would like to thank Jian Wu, Yuanyuan Zeng, Eugene Chai, Buyoung Yun, and Xinyu Zhang for not just sharing the office space, but also being friendly and witty conversation partners on just about any topic. I would also like to thank my colleagues Hyoil Kim, Alexander Min, and Jaehyuk Choi for their support and encouragement.

I would like to thank my friends Se Young Chun and Taeho Kgil for looking out for me for such a long time. I also want to thank Professor Leslie Olsen and Ms. Robin Fowler at Technical Communication. Finally, I would like to thank my family for their unconditional love and support.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

Appendix

CHAPTER I

INTRODUCTION

1.1 Motivation

With a long history, security in computer systems is a major topic in computer science. Recently, however, the Digital Revolution has brought significant challenges to the domain of computer security. Fueled by the multitude of inexpensive high-performance computing devices and ubiquity of high-speed networks, people have come to rely heavily on computers for their needs for processing and communicating information.

Inevitable to this megatrend is the risk of using computers for handling sensitive information. Computers are now heavily used not only for organizational classified information, but also for privacy-sensitive information of individuals such as ID/password tokens, bank account numbers, credit card numbers, or Web browsing history. As this trend is expected to continue, the risk of exposure of such sensitive information is also increasing.

Exacerbating this situation is the unprecedented level of seriousness of a security breach: due to the dramatic advancements on storage capacity and communication throughput, a large amount of sensitive information can be stolen via an extremely small device such as a high-capacity thumb drive, or can be leaked almost instantly through a broadband network. Should such a breach occur, it is extremely hard to contain the damage.

In addition, we are now living in a world of hostile computing environments laden with a plethora of malwares and various forms of online attacks [40, 74]. We have already entered into the world where one should always presume an unverified or unprotected system to be hostile or possibly compromised. It is also fairly reasonable to imagine that an operating system, which runs with privilege, had been subverted by malware at some point in the past and became an agent of the intruder. Unfortunately, the task of detecting such a breach, let alone fixing it, is a daunting task because the intruder can easily hide itself using, for example, kernel rootkits.

The new challenges in computer security, however, seem not to be overcome, at least significantly, by the current solutions on information protection in computer systems. Attacks and exploits have become sophisticated and scalable whereas defense against them has been either a patchwork or an incremental improvement on existing solutions. As computer systems are getting bigger and more complex, opportunities for new exploits are increasing but existing defense tactics are mostly reactive. The consequence of a system compromise is becoming serious but current systems do not provide further protection after a compromise.

To be sure, there is no shortage of cryptographic algorithms, security policies, and software verification tools, all of which are the fundamental building blocks for constructing a secure system. Cryptography, using solid mathematical results, provides strong methods for secure delivery and storage of information. Security policies deal with how to specify rules regarding access control. Verification tools offer systematic solutions to finding software defects. Nevertheless, we notice that many real world problems are actually caused not by the immaturity or insecurity of those fundamentals, but by the way in which they are put together.

Many real world problems arise because of the insecurity in the trust base assumed by a given system. Trust base of a system is defined as a collection of hardware and software that the security depends on and therefore must be trusted unconditionally. Trust base has the property that its compromise leads to a total collapse of the security of the system. Therefore, it is highly desirable to have a smaller, simpler trust base in order to minimize the chance of a compromise in the trust base.

In this respect, much trouble is caused by the current practice of taking the operating system in its entirety as the trust base by most applications. This practice has been widely accepted and rarely challenged because the operating system, which manages hardware resources, has to run with privilege and thus has to be trusted. But today's operating systems have become difficult to secure because of their large size and wide interface. We can now no longer accept the trustworthiness of operating systems unquestioned.

In addition, the privilege granted to the operating system seems excessive in that it is capable of reading and writing any application data. If malware infects a system and the operating system is compromised, the malware can essentially spy on everything in the system. This is because the operating system is under the control of the malware, and the operating system is given the privilege to access any data in the system.

This privilege is therefore at odds with security concerns for many applications. These applications are not just the ones that handle highly classified information, but also the ones that ordinary people use every day. For example, a bank customer making transactions using a web browser should be concerned about the secrecy of banking information. Also, a video game developer should be concerned about protecting the game content against digital piracy. Unfortunately however, there is no way to protect both of the applications against a malicious operating system determined to intrude and spy on the applications.

In this thesis, we posit a system architecture that separates information protection from resource management. Information protection, as in data secrecy and integrity, is provided by a new security protection layer. Resource management, as in making resources available, is provided by the operating system in the traditional sense. The new protection layer is made more privileged than the operating system so that an untrusted operating system cannot undermine the secrecy and integrity of information. This separation allows a user application to selectively take its trust base according to its own security need. For its need for secrecy and integrity, the user application only has to trust the new security layer, not the operating system.

This thesis demonstrates the feasibility and effectiveness of separating protection from management by implementing our design in a real system. We explore the protection model based on cryptography and extension on paged memory, an implementation of the protection model using a virtual machine monitor, and the impact of the protection on application programming and operating system. This thesis concludes that the information secrecy protection can be effectively separated out from the purview of traditional operating systems.

1.2 A case for separating information protection from management

Making a case for separating information protection from management starts with this observation: user applications, not operating systems, are the ultimate consumer of information. When we consider computing as information processing, user applications are the ones that draw meanings from data, perform computations, and produce results. On the other hand, operating systems, although managing resources that contain the information, do not need the actual information in the resources.

However, current operating systems are capable of accessing any information because of the way the privilege is given to the operating systems. Therefore, when it comes to the secrecy or integrity of applications' data, applications just have to trust the operating systems.

Given the insecurity in today's operating systems, we need a new protection paradigm that reflects the abovementioned observation: instead of trusting the operating systems, we remove the need to trust by making the operating systems unable to access application information.

This can be made possible by placing a new security layer running with higher privilege than the operating system. This layer protects user applications by cryptographically securing the information contained in the resources. The operating system manages the re-

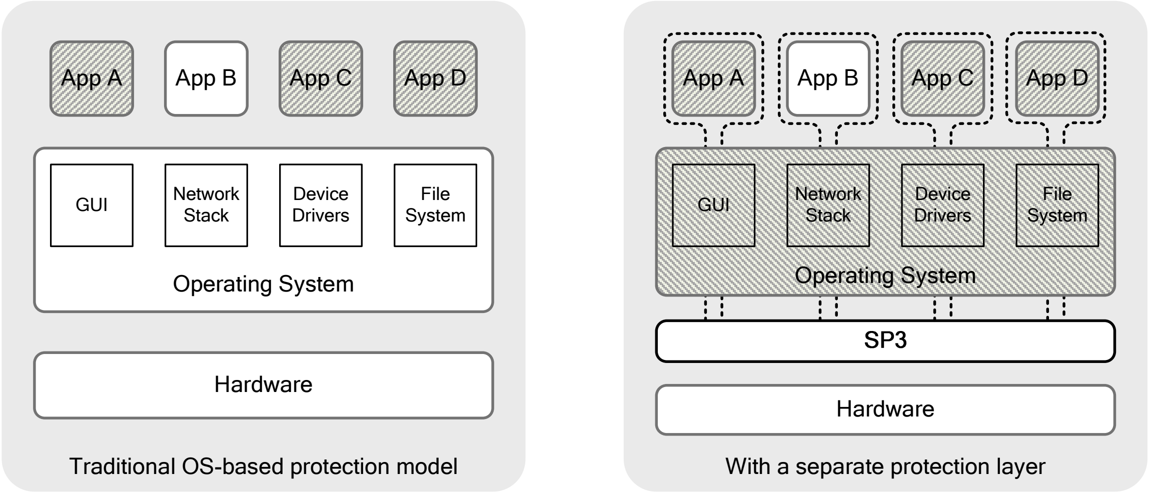


Figure 1.1: A new layer of application protection system inserted between hardware and operating system. The dark shade indicates the components that are not included in the trust base of application B. In the conventional model, the whole operating system must be trusted whereas the operating system does not need to be trusted anymore with the new model. In the new model, the application B only has to trust the new protection system.

sources as usual, but applications do not have to trust the operating system for data secrecy and integrity.

Figure 1.1 shows the computer system structure and its trust relationships. The figure contrasts the traditional model, shown on the left, with the new model, on the right, in which a new security layer - labeled as SP^3– is inserted between operating system and hardware. The dark shaded units are the components that are not part of the trust base of application B. In the traditional model, application B has to trust the operating system. Due to the isolation provided by the operating system, the other applications are excluded in the trust base of application B. In the new model with the SP^3 layer, the trust base of application B now excludes the entire operating system. The dotted line wrapping the applications signifies that these applications are protected directly by the SP^3 layer. In this thesis work, information secrecy protection is provided by a system called *SP*^{3}, hence the name SP^3 for the layer.

It is important to note that the new layer only protects the secrecy and integrity of the information and not the availability of the resource. This means that if availability guarantee is part of the application's security requirements, then the application still has to trust the operating system.

It should be emphasized that providing availability to the applications is in fact the raison d'etre of the operating system. We know that the original purpose of an operating system is to manage hardware resources and make them available to user applications. The operating system absorbs the differences in the underlying hardware and provides application programs a consistent execution environment. The operating system also multiplexes hardware resources, i.e., CPU time and memory, to multiple application programs.

Security properties	Traditional model	SP^3
Information Secrecy	Operating system	SP^3
Resource Availability	Operating system	Operating system

Table 1.1: Security properties of user applications and the system entities that should be trusted.

Table 1.1 summarizes the security properties sought by user applications and the corresponding system entity that is responsible, hence must be trusted, for each of the properties. In the traditional model, the operating system should always be trusted by user applications. This means that if the operating system is compromised, the secrecy and integrity of an application cannot be protected. With SP^3, applications trust the operating system only for resource availability. For information secrecy, applications trust SP^3. This means that even if the operating system is compromised, applications still have secrecy protection.

In this thesis, we show that the SP^3 can be implemented as a small and efficient layer, affirming that the benefit of separating protection from management outweighs the overhead of the new layer. We show that SP^3 can be implemented either by hardware modification or by extending a hypervisor, which is significantly smaller than conventional operating systems, while incurring little overhead.

In a sense, the removal of the operating system from the trust base of user applications can be likened to the removal of computer network from the trust base of end hosts. In

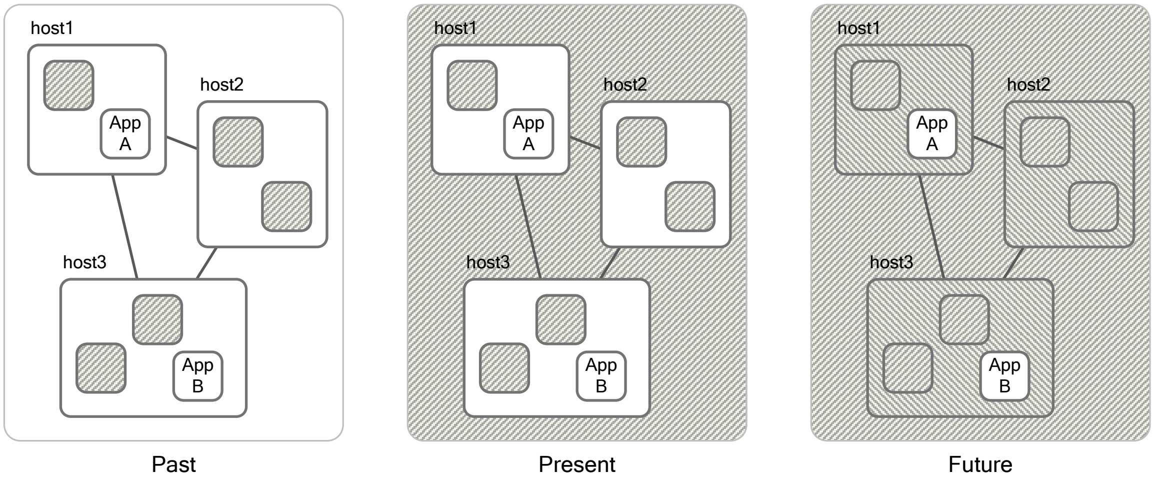


Figure 1.2: The reduction in the trust base assumed by applications as the computing environment evolves. In the past, when computers were connected with trusted medium, trust base of applications A and B included the network and the hosts. In the present, A and B trust their host computers, but not the network. In the future, A and B do not even trust the host computers, which is made possible by SP^3.

the old days before the advent of network security, it made sense to just assume computers were connected with a trusted medium. Therefore, under this assumption, programs did not have to secure their communication. Programs such as `telnet` and `rsh` are such examples. When we cannot trust the medium any more, as is our current computing environment, we have to secure the host-to-host communication by establishing connections using cryptography-based solutions such as `ssh`. In effect, the use of cryptography removes the computer network from the trust base of end hosts. However, the end applications in each host still have to trust their operating systems. If we cannot trust the operating systems any more, as is happening now, we have to secure the applications by removing the operating system from the trust base of applications. SP^3 can be considered as a step-forward in this direction.

To illustrate this point, Figure 1.2 shows the trust base that has to be assumed by applications A and B. In the past, shown on the left, the applications had to assume the network and the hosts are trusted. This assumption was justified at that time given the size, complexity, and manageability of the network. In the present day, shown in the middle, the network is removed from the trust base of the applications. This is made possible by the tools that can establish secure connections. In the future, shown on the right, the applications do not have to trust their operating systems with the help of SP^3, which can remove the operating system from the trust base of the applications.

1.3 Comparison with related systems

The phrase ‘separation of protection from management’ is already a familiar expression in the systems software literature. One of the uses of the phrase has to do with 90’s microkernel research, which promoted modular and layered operating system kernel design [7, 37, 68]. In this design, a traditional monolithic kernel is replaced by a much leaner microkernel, which only implements primitive abstractions such as address space and interprocess communication. Higher-level constructs such as memory manager, file system, and schedulers then form a layer that runs on top of the microkernel. One of the motivations of microkernels was the potential benefits resulting from the fact that a microkernel enables the use of specialized versions of operating systems. This is because the specifics of management algorithms can be separated out from the core microkernel. For instance, the Exokernel [37] is capable of running multiple custom ‘library operating systems’ on top of the microkernel.

The meaning of ‘separation of protection from management’ in the context of microkernels is therefore very different from the one argued for in this thesis. In the microkernel, protection really means ensuring safety in allocating and reclaiming resources. Microkernel has to retain the power to always revoke and reclaim resources allocated to the library operating systems. The general management policies are separated out and implemented by library operating systems, whereas the ability to safely multiplex underlying resources is retained and protected in the microkernel. On the other hand, our meaning of protection is about ensuring information secrecy and integrity.

There are two closely related systems that are in line with the theme of this thesis and here we compare them with our SP^3 system. One is Overshadow [28] of VMware [97, 107] and the other is Xom/Xomos [66, 67]. Overshadow is a hypervisor-based security solution that can protect the secrecy and integrity of user applications without modifications. Xom is a secure hardware architecture that can execute encrypted code in the memory. Xomos is the untrusted operating system that runs on top of the Xom architecture. SP^3 shares a general objective with these systems, but differs in specific design choices thus resulting in a different feature set.

	Feature	SP^3	Overshadow	Xomos
1	Secrecy protection against untrusted OS	✓	✓	✓
2	Integrity protection against untrusted OS		✓	✓
3	Feasibility of hypervisor-based solution	✓	✓	
4	Feasibility of hardware-only solution	✓		✓
5	Integrated key management	✓		✓
6	API/ABI for applications	✓		✓
7	Support for unmodified OS/applications		✓	
8	Support for unmodified compiler/linker	✓	✓	

Table 1.2: Comparison of SP^3 with related systems.

Table 1.2 summarizes the features and characteristics of the three systems. The differences between Overshadow and SP^3 are caused by one differing design goal: Overshadow targets unmodified operating systems and applications to retrofit security to existing applications, whereas SP^3 provides a secure execution environment that applications can exploit by programming explicitly. This results in the differences in lines 6 and 7. Xom/Xomos is inherently hardware-oriented; thus it requires special hardware (3,4), a special operating system (7), and a special compiler (8).

1.4 Thesis contributions

This thesis demonstrates the feasibility and effectiveness of creating a new secrecy protection layer for applications .The contributions of the thesis are summarized as follows.

- *Design of a page-granular memory secrecy protection system*: We proposed a new notion of protection domain for information secrecy. User-level applications executing within their protection domains can conceal their memory contents on a per-page basis against the host operating system. An entity more privileged than the operating system implements and enforces the protection so that the conventional operating system can be removed from the trusted base of the user applications regarding information secrecy.

- *Efficient realization of the protection with a hypervisor*: We realized the protection system by modifying a hypervisor. The underlying hardware is abstractly extended so that the hardware directly supports the protection system, and then the extended hardware is emulated using a hypervisor. To enhance performance, we implemented optimizations that reduce costly cryptographic operations.

- *Thorough reexamination of system software constructs under the secrecy protection*: We investigated the implication of the new secrecy protection on the system software and application programming environment. A wide range of software constructs has been constructed under the implicit assumption of the trusted operating system, and therefore must be revisited with the new secrecy protection system. We studied and resolved issues with the application programming environment and the traditional operating system implementation.

- *Using the protection to guard multimedia contents and media player*: We tackled the problem of securing a trusted media player program that is to run on the computer of an untrusted end-user. Using the application secrecy protection, our system can secure the input multimedia file, decoded multimedia output, and the media player program itself from reverse-engineering. Our system presents a practical and flexible system solution for digital rights management that can satisfy both the copyright holders and the end-users.

1.5 Scope and limitations

In this section, we delineate the scope of the dissertation and clarify the limitations of the thesis work.

1.5.1 Scope

First, this thesis primarily focuses on the architecture side of system software organization for information secrecy protection. This means that we do not delve heavily into cryptography primitives or access control methodologies. Instead, we treat them as our building blocks and we take the usual assumption that they are secure. Among the security properties, providing protections for integrity and availability are left out of the scope. There are already many integrity protection solutions and implementing a mechanism for hard-integrity protection is rather straightforward. We consider that providing availability is the purpose of the operating system, and separating availability part from the operating system beats our original purpose. Therefore, the problem of providing availability, such as guaranteed quality of system services, is out of the scope of the thesis.

Second, this thesis mainly engages in providing mechanisms for reducing trust base by defining hardware or software interfaces. We do not study the subject of how to verify, establish, and maintain trust relationships. In the same vein, we do not study policy-side issues such as identifying the criteria by which the trust base can separated. Analyzing and controlling information flow in software is an important but not a pertinent topic in the thesis.

Third, this thesis concentrates on memory resources for information secrecy protection. We do not provide comprehensive secrecy protection to other resources such as file storage or network. Although our system can provide transparent secrecy protection to the user data of stored files, one should use separate secure file systems if he wants to secure the file system metadata. For secure network communication, one has to rely on secure communication protocols, which is easy to do.

Last, this thesis only provides means to user application programs to protect themselves. We do not attempt to provide an external mechanism to secure originally insecure user applications. Finding bugs and vulnerabilities in software is important but irrelevant to the theme of the thesis. This thesis is not about how to defend against specific types of threat or to resolve specific security vulnerabilities.

1.5.2 Limitations

The limitations of the thesis work largely originate from our design decisions on constructing a new protection system that allows untrusted operating system. There are two fairly independent design spectrums from which we have to make trade-off decisions that can impact the size and complexity of the new protection system itself: one is the feature set size of the new protection system, and the other one is the level of intrusion of the protection system to the existing system. We preferred and settled on the design choices that can yield a small, neat, and simple system. For the former, we focus on memory secrecy protection, reducing the size of the trust base. For the latter, we sacrifice a small degree of transparency, simplifying the logic of the trust base. As a result of such design decisions, the thesis work has the following limitations.

First, our current implementation lacks an active-form of information integrity protection. Active-form of integrity protection means that the protection system proactively checks the hash-sum of the data before use and enforces rules that prevent unauthorized modifications on the data. Implementing such a system is relatively easy and straightforward. We sketch a design of integrity protection in a section of Chapter VI. Note that a passive-form of information integrity protection is provided as a side-effect of information secrecy protection. In the current implementation, any corruption – even just one bit – on a protected memory page results in the corruption of the entire page due to the avalanche effect of block cipher, leading to a crash or failure of the program execution.

Second, our information secrecy protection is limited to the memory resources. Although securing memory has a far-reaching effect, such as transforming conventional file system into encrypting file system, other system resources such as network I/O must be secured using conventional secure communication protocols. Applications that require more than a simple encrypting file system must use other secure file system implementations.

Third, largely unstudied is the potential security issues with malicious operating systems leveraging the fact that user applications fundamentally have to rely on the operating system for core services such as timer, synchronization, or identifier management. By misrepresenting such services, protected applications can be coaxed to leak some information to the untrusted operating system.

Last, we require source code modifications on traditional operating systems and applications written previously under the assumption of trusted operating system. Although we consider that the exposure of the protection system is necessary and the intrusion to existing systems is minimal, there could be cases where source code modification is impossible. For example, someone may want to execute a legacy application binary compiled from the source code written with trusted operating system assumption, and he may not possess the source code. However, we observe that the people who have interest in protecting their application programs usually have access to the application source codes. Therefore, we believe the sacrifice in transparency would not have great implication in the practicality of our solution.

1.6 Literature survey on computer system security

Computer security is a very broad term, encompassing many domains of security researches. To better illuminate where this thesis stands, this section is devoted to a literature survey on relevant security research domains.

1.6.1 Theory – building blocks of a secure system

Cryptography is the first and foremost building block of constructing a secure system. Usually, programmers treat cryptography as a set of given functions from some standards such as AES [76], and that approach is often sufficient. Aside from cryptography, computer security theory includes the fields of protection model, access control, formal method, and code obfuscation. For example, Harrison [48] defined a theoretical model of protection in operating system and discussed the safety problem of protection systems. Role-Based Access Control (RBAC) [38] is an example that represents the results on the theory of access control. Spi Calculus [6] is a way to formally verify security protocols. Code obfuscation refers to a systematic technique that intentionally makes it very hard for a reader to infer the program logic from the source code. Theoreticians investigated on the limits on the effectiveness of obfuscation: Barak [11] showed the impossibility of obfuscating a program. In [9], it is proved that de-obfuscation problem is in NP.

This thesis does not investigate any security theory listed above. Rather, like many other systems work, we rely on the result of such theoretical works and use them to build a secure system.

1.6.2 Classic – protection rings

The concept of protection rings has been the lynchpin of the fundamental facility for enforcing protection in modern computer systems. For example, the usual division of kernel and user relies on the protection rings of a processor with varying privileges. Lampson [65] laid out the first framework for protection and access control in computer systems. Saltzer's study on the protection and sharing in Multics [86] introduces the principle of the least privilege. Saltzer also helped to shape the modern structure of protection mechanisms in computing systems [87]. Recently, attempts were made to provide fine-grained protection to the kernel by utilizing unused protection rings and segmentation facility of x86 architecture, which can be found in the works of Chiueh [29] and Witchel [109]. Another important use of protection ring other than user/kernel separation is to insert the layer of hypervisor below the operating system.

This thesis utilizes the protection rings in order to construct the home layer of the new protection system that must be more privileged than the operating system. However, the secrecy protection implemented by the layer creates a protection domain that is orthogonal to the existing protection ring concept.

1.6.3 Trust – root of belief

As reflected from Thompson's Turing Award lecture [102], any software system may just have to unconditionally trust some parts of hardware, software, compiler, and ultimately the people who create them as prerequisite. The Root of Trust therefore refers to the seminal trust base from which other trust relationships are established.

Yee's work on implementation of a secure computing platform [111, 112] explored the use of a secure coprocessor hardware, which serves as the Root of Trust, for realizing a trusted computing platform. Trusted Computing Group's Trusted Platform Module (TPM) [103] is a standardized Root of Trust developed by the computer industry. TPM is capable of establishing and managing trust among computer hardware and software, and is typically used for secure bootstrapping. Other industry efforts in this direction includes Intel's LaGrande [55] and Microsoft's NGSCB [72]. For a notable example of using TPM for secure bootstrapping, Kauer's OSLO [60] implemented a practical secure boot loader which improves security of Trusted Computing.

The trust base of our solution includes the hardware and the protection layer implementation (i.e., the hypervisor). Therefore, they must be securely bootstrapped and we can utilize TPM and OSLO to achieve this. However, we do not have to rely on TPM or Trusted Computing-style integrity verification to enforce the information secrecy protection.

Reducing size and complexity of Trusted Computing Base (TCB) are the themes of Hohmuth's work [51] and Singaravelu's AppCore [96]. Hohmuth precisely defines the notion of trust in system software with respect to different security properties that the trusted system can guarantee, clarifying what it means by ‘trusted’ or ‘untrusted’ depending on the security property. These works are done by the people of L4 research community in Europe. Specifically, Hermann Härtig utilized his early L4Linux work [50] to realize his design of reduced TCB, the Nizza [49] architecture. The Nizza architecture influenced the Hohmuth and Singarevelu’s works. Singarevelu’s AppCore [96] takes one step further from just reducing TCB of operating system to extracting security-sensitive portions of applications, called AppCore.

IBM’s PERSEUS system architecture [79] is conceptually similar to Nizza security architecture. The PERSEUS is strongly motivated by observing why direct applications and using secure modules alone are bound to fail without properly coordinated system security structure. It assesses practicality of the system in terms of the usability of the system, which is directly related to the ability of having secure system without much modification on the implementation or interface.

Compared to these works for reducing TCB, our system takes a fundamentally different approach: instead of changing the internal structure of the kernel, we create a new protection layer that is orthogonal to the existing system constructs, therefore, inflicting minimal changes to the internal structure.

Establishing a trusted path to an end-user is also important in order to guard against Trojan horses. To secure the trusted path of monitor screen, a trusted window system [94] is implemented on EROS operating system that reserves a portion of display for direct communication with the end-user.

Although our system does not directly address the trusted path problem, our system can easily create a trusted path to the end-user by means of an encryption-enabled I/O device, which is demonstrated in Chapter V.

1.6.4 Integrity – tamper-proofing software contents

It is important to protect the integrity of software from potential malicious modification attempts, and to do so requires tamper-proofing techniques for software contents in both hardware level and software level. In the hardware level integrity protection, Collberg [30, 31] studied various strategies for providing hardware-based software integrity protection. Secure processors – processors with cryptography-enhanced hardware - usually have the functionality for checking the cryptographic hash of the main memory contents. MIT's Cerium [27] and Kgil's ChipLock [61] are the instances of such secure processor architecture. Lie's XOM/XOMOS [66, 67] can also be considered as the first secure processor architecture that can host a fully untrusted operating system, along with physical tamper-proof and encryption of main memory contents. Suh's AEGIS architecture [98, 99] details how to physically provide main memory integrity protection.

In the software-level integrity protection, privileged software in a computer system can implement software-based integrity verification and enforcement mechanisms. For example, Seshadri's Pinoeer architecture [91] and Park's PIV architecture [77] provide solutions to remotely verify the software contents and to enforce integrity of software codes.

In this thesis, physical memory integrity protection is largely out of the scope. However, physical tamper-proof techniques must be employed if we need to guard against physical attacks on the main memory or processor hardware. Active-form of software-level integrity protection is also not in the scope of this thesis, and left as a future work.

1.6.5 Isolation – containing damages

A very effective security strategy is to put in place an isolation scheme that prevents a failure or a compromise in a system from damaging others. For example, Goldberg [45] demonstrates a way to confine untrusted helper applications such as plug-in modules. Interposing on system calls, Mitchem's kernel "hypervisor" [73] isolates and contains damages. Sekar [89] proposed a software isolation scheme called "model-carrying code" that can safely execute untrusted applications.

Our system differs in several ways from the abovementioned isolation schemes. First, our system solves a different problem: our goal is to protect trusted applications against untrusted operating system, which is basically the inversion of the goal of the traditional isolation schemes. Second, our system does not restrict the functional capability of applications or operating systems, whereas traditional isolation schemes try to confine applications. Third, our system allows applications to run within the full context of the operating system, but isolation schemes tend to put applications in a virtual environment that is limited in many ways.

Partitioning a program in terms of importance is another effective way to isolate critical and privileged code. For example, Brumley's Privtran [23] is an automated tool that can partition a program into sensitive or important parts and the rests. It uses annotations on the source code specified by programmers.

User applications running in our system can potentially benefit from the techniques for tracking down the use of sensitive information in a program because in our system, applications have to identify and annotate the memory to share with the operating system.

Recently introduced late-launch feature in the commodity processor enables running a completely isolated instance of a verified piece of code in parallel with an already executing host operating system. For instance, McCune's Flicker [71] presents an infrastructure for executing security sensitive code in complete isolation while trusting very small code.

Our system significantly differs from Flicker since our system does not create a separate, isolated machine instance. Instead, our system allows a protected application running as a legitimate process of a host operating system.

1.6.6 Robustness – reducing vulnerability and promoting safety

Many bugs and defects affecting information security can be detected by analyzing information flow. Myers [75] presented a decentralized model for controlling information flow in systems with mutual distrust and decentralized authority. Later, Zdancewic [114] presented a technique, called secure program partitioning, for protecting confidential data in distributed systems containing mutually untrusted hosts by annotating programs with security types that constrain information flow.

Due to the importance of the code running in the kernel space, many efforts are devoted to produce quality kernel code. For example, Petroni designed a system [78] that automatically detect kernel control-flow attacks by dynamically monitoring operating system kernel integrity, which based on a property called state-based control-flow integrity.

Many efforts are made to improve the performance and accuracy of bug-finding tools. For recent instances, EXE [26] took advantages of both static and dynamic analysis. Kremenek's code analysis tool [64] infers specification in the code. Qui's application level toolkit [83] helps enhance robustness of application code against Denial-of-Service (DoS) attacks by systematically injecting protection mechanisms into the code.

To reduce the chance of writing a defective code, safe programming languages and practices are promoted. Cyclone project [58, 100], for example, enhances C language to introduce strong type-safety in writing systems code. Microsoft's open source Singularity operating system [53] relies entirely on the safe language the operating system is written in.

As a security extension to Dynamo [10] and RIO [20] dynamic translation and instrumentation framework, Kiriansky proposed a way to safeguard the path a program chooses at runtime, called program shepherding [63].

Although finding bugs and vulnerabilities is not among the goals of this thesis research, our system implementation can greatly benefit from the aforementioned techniques for enhancing robustness, resulting in a more secure and trustworthy trust base.

1.6.7 Tactic – defense against specific threats

Exploiting buffer overflow has been a major technique that allows intruder to run a foreign code in a system. By overflowing the buffer in the process's stack – stack smashing – attacker can override the return address of the function to alter the victim process's execution path. Defense against stack smashing attack [12] has been proposed. Recent Intel's NX flag extension [56] prevents primitive forms of stack smashing attacks. An address randomization technique that lowers the success rate of buffer overflow attacks, called Address-Space Layout Randomization (ASLR), further provides defense against buffer overflow attacks, despite its own limitations [92].

Hiding information without help from higher-level constructs has both theoretical and practical interests. Linn [69] proposed obfuscation of executable code to improve resistance to static disassembly. Popov used signals to facilitate binary obfuscation [81]. Tsafrir [104] discusses a particular exploit in which one can monopolize CPU share without super-user privileges.

File systems draw particular attentions for security research due to the fact that the file systems deal with massive amount of persistent data storages. There are many versions of secure file system. For example, early classical encryption file systems include Zadok's Cryptfs [113] and Blaze's crypto file system [18]. Zhao's SVFS [115] uses hypervisor to enable an untrusted file storage.

1.6.8 Makeover – new operating system designs

Because of the perceived insecurity of the major operating systems and the difficulties of retrofitting current operating systems [57], many attempts have been made to overhaul the operating system design and start from the scratch to build secure operating systems.

Although not directly motivated by the security concerns, microkernels gave us insights on what is fundamental and what is incidental in operating systems. Mach microkernel is introduced in Accetta's 1986 paper [7]. Dividing the process abstraction into two orthogonal abstractions of task and thread is among the novelties of Mach. Exokernel [37] is a microkernel architecture that focuses on safe and secure multiplexing of resource among application-level library operating systems. Liedke's L4 [68] gives insights about what operating system constructs are fundamental. In this work, Liedke derived a minimal and complete set of functionalities that can be used to construct any operating system features. Ford [39] discusses the construction of recursively structured microkernel.

Attempts to create a new operating system with fresh design have been made consistently. Operating systems such as SPIN [17] and EROS [93, 95] were designed to support high-degree of extensibility along with infrastructures for run-time verification. Asbestos operating system [36] uses label for system calls to facilitate control of information across system level. Microsoft Singularity project [53] is specifically designed with security and safety in mind, heavily relying on safe language.

One use is to remove operating system from the trusted computing base. Litty's Manitou [70] modifies Xen to provide memory integrity protection to users.

1.6.9 Virtualization – one ring to bring them all

Recent resurgence of virtualization has tremendous impact on the system security research because hypervisors serve as an efficient and non-intrusive implementation point for many security ideas. For example, Ta-Min's Proxos [101] allows applications to configure their trust in the operating system by partitioning the system call interface into trusted and untrusted components. Proxos uses hypervisor (Xen) to host two operating systems; one is to run untrusted operating system and an application, and the other one is to run trusted operating system to which the application request sensitive services. Customizing trusted computing standards has been explored in Terra [43]. vTPM work [16] of IBM virtualizes the TPM module in a virtual machine monitor.

In the Potemkin virtual honeyfarm system [106], Vrable and Savage considered that the rapid evolution of large-scale worms, viruses and botnets have made Internet malware a pressing concern. Such infections are at the root of modern scourges including DDoS extortion, on-line identity theft, SPAM, phishing, and piracy. They have built a honeyfarm system based on virtual machines to emulate over 64,000 internet honeypots.

Chen's Overshadow [28] uses a hypervisor to provide security to user-level applications without trusting the operating system. Overshadow achieves this without modifications on operating system and user-level applications. Ports [82] further discusses future direction and outlook in the context of Overshadow project. It details the challenges and requirements for the application in the context of untrusted operating system.

Hypervisors have been used to implement fault-tolerance schemes. Bressound [19] built a fault-tolerant system based on a hypervisor. Based on the Xen replication, Cully's Remus system [34] implements a high-availability system targeted for disaster recovery solutions. Remus achieves high performance and low latency by using a speculative method.

1.7 Thesis outline

Chapter II proposes the *Software-Privacy Preserving Platform* (SP^3) for secrecy protection. The SP^3 protection model is defined as an extension made to the hardware architecture of an abstract CPU with a paging unit. This includes the core design rationales, application execution scenarios, and implementation alternatives.

Chapter III presents a *hypervisor-based implementation* of the SP^3 protection model. The extended architecture is efficiently emulated by the modified hypervisor which is executing between operating system and hardware. We implement the system by modifying the x86 architecture, Xen hypervisor, and the Linux kernel. Evaluation results show that the hypervisor-based implementation incurs small performance overhead.

Chapter IV deals with the impact of SP^3 on the *programming environment* of both application and operating system. We detail the implications of the SP^3 memory access rules on the application programming, building and loading. We address these application-level issues by code annotation, object file extension and binary loader modification. We then discuss operating system issues in memory manager and signal in an SP^3 environment. We address these issues by modifying the Linux kernel.

Chapter V presents an important use case of SP^3: a *multimedia player* that enforces copy protection. Utilizing SP^3 protection, rights-enforced multimedia contents are secured. We also demonstrate how SP^3 can secure the codec and media player itself, the weakest point of the rights-management infrastructure.

Chapter VI concludes this thesis.

CHAPTER II

INFORMATION SECRECY AND SP3 ARCHITECTURE

2.1 Introduction

Privacy is perhaps the most important security concern for many user-level applications; users would prefer to crash their applications, rather than revealing their private data. This is because exposing sensitive information is considered worse than failing to execute applications. In fact, except for safety-critical real-time systems, most applications and users can cope with execution failures with various recovery and backup strategies. For instance, a crashed word processor program can recover from a backup copy, or a disconnected `ssh` client can simply reconnect. However, the exposure of sensitive information in these applications is considered far worse than their execution failure. The word file may contain classified or confidential information, and the exposure of the private key of the `ssh` client amounts to a compromise of the entire communication.

Privacy[1] protection for applications is traditionally provided by operating systems through process isolation. Ironically, however, operating systems themselves can be the biggest threat to the applications; a modern operating system is very difficult to secure because of its extreme size and wide interface. Once compromised, the operating system becomes a powerful weapon for thieves of applications' sensitive information. We argue that this

[1] In this thesis, 'privacy', 'secrecy', and 'confidentiality' refer to the same security property and are thus interchangeable.

problem can be solved by devising a new application protection system that is not under the operating system's control and can thus directly secure the information of applications. Even if the operating system were compromised, the protection system can prevent the exposure of application information. The worst an attacker can do is to crash the system, thus causing the applications to fail to execute, which is usually less harmful than exposing privacy information.

Designing such a protection system, however, is a difficult task, especially if it is to be practical. Any practical protection system must be simple, general and easy to use. Moreover, to be practical, a protection system should require minimal or no changes to the software environment so that it can be applied or deployed easily to existing systems. We have designed a protection system that can directly secure the application information. This protection system, which we call *Software-Privacy Preserving Platform* (*SP*^{3}), provides a simple and general interface, requires only minimal changes to existing systems, and creates an easy-to-deploy trust base. We introduce *software-privacy* as a new protection measure that SP^3 guarantees to applications by encrypting the contents of memory. SP^3 guarantees that a correctly-written application can protect the secrecy of its memory contents.

SP^3 chooses the *information* contained in the memory, not the memory itself, as the target of protection. In addition, it uses a memory *page* as the unit of protection, which is also the unit of memory management. These design choices make it easy to separate protection from management without sacrificing flexibility and generality. SP^3 ensures that an operating system sees only encrypted images of the pages that are private to applications. The operating system, which is usually indifferent to the user's memory content, is not obstructed in managing memory with these encrypted images; that is, protection is "orthogonal" to management. SP^3 achieves a high degree of protection orthogonality, which also allows data secrecy to be an intrinsic property of operating system services. For example, SP^3 turns an unmodified regular file system into a secure one that can write encrypted data to disk.

SP^3 extends the conventional paging system, so an application process sees different memory contents in its virtual address space depending on its execution context. SP^3 guarantees the correctness and safety of the protection mechanism by carefully controlling the keys used to encrypt memory pages: the actual encryption keys are hidden from the operating system. The operating system is only allowed to *associate* each page with a key. Permissions to the keys are completely controlled by individual applications, *not* by the operating system.

2.2 Motivation

We motivate this work by reviewing problems of current practice in protecting application information (Section 2.2.1), and then discussing challenges in creating a new protection system for existing computing systems (Section 2.2.2).

2.2.1 Problem

The operating system is responsible for providing an execution environment for user applications. It also provides data privacy of applications through process isolation in which each process is protected from other processes. This requires the applications to trust the operating system, and therefore, the system security depends on how secure the operating system is. The operating system is thus strongly secured and the operating system kernel runs with privilege, wielding unlimited power.

Unfortunately, from the perspective of user applications, a large part of operating system code must be trusted. Moreover, many parts of the trust base, such as core kernel and device drivers, run with privilege. For these reasons, relying on the operating system for application data secrecy has the following problems.

First, there is no effective second-line of defense that applications can resort to in case the operating system is compromised. Many applications need to protect sensitive information, but once an attacker seizes control of the operating system, it is very easy for her to steal information from the application. Any effort to protect the information will be futile as the attacker can exploit the operating system's omnipotence to subvert, reverse-engineer, or simply disable the protection mechanism.

Second, it is very difficult to guarantee the operating system's trustworthiness. Verifying the correctness of an operating system has become intractable as its size and functionality continuously grow to meet the increasing demand for more functionalities and features. Furthermore, the operating system is increasingly built with components from diverse sources.

Third, many negative social and technical side effects arise when someone forces this type of protection by, for instance, using trusted computing. The biggest problem of the current trusted computing approach is that it severely limits the users' freedom to choose and install operating systems or programs, regardless of whether they are certified or not.

These problems can be solved by a new protection system that is not under the operating system's control and that can directly secure the applications' information. This protection system can always guarantee the data privacy of applications even when the operating system is compromised. Therefore, it serves as an effective second-line of defense for applications. For the same reason, the trustworthiness of an operating system does not have to be verified; a malicious or faulty operating system may fail the execution of applications, but it cannot steal or divulge the applications' information. Since the new protection system runs independently, users can choose any operating system and program to run without risking the exposure of sensitive information.

2.2.2 Challenge

In general, creating a practical protection system is a challenging task. A careful design is required to create a protection system that is simple, general and easy to use. Moreover, to create a protection system that directly protects the information of applications, we must overcome the following two challenges. First, we must preserve the operating system's usual management power, incurring minimal changes to existing systems. The new protection system may restrict the operating system by enforcing certain rules and hence preventing the operating system from performing operations against the rules. However, the restrictions should not obstruct the operating system from performing legitimate management jobs. Moreover, an existing operating system may not be compatible with the new rules, thus requiring adaptation. The adaptation should entail minimal changes to the operating system. Second, we must find an implementation that is small and simple to verify. With the new protection, the operating system can be verified less stringently, since applications can still be protected even when the operating system fails (as a result of its compromise). However, the mechanism that implements the new protection should be fully trusted, and hence, the correctness of the implemented protection is critical to the security of the entire system.

2.3 Design objectives

Practicality is our primary concern in creating a new protection system. It is usually difficult to design a protection system since its usability depends greatly on the simplicity and generality of the protection. Many past and current protection systems with complex interfaces failed to survive, or are simply not used. For instance, although it provides more features and fine-grained control, segmentation has not been chosen for memory protection over the simpler, easier-to-use paging system. A protection system must, therefore, be simple and general so that its benefits can outweigh the accompanying inconvenience and overhead.

We have another practicality-related goal: the protection system must be easy to apply to existing systems. Any required modifications to an existing system must be 'orthogonal' extensions so that the system may preserve its original function and structure, and the extensions should not restrict the users.

Therefore, we take *simplicity*, *generality* and *orthogonality* as our design objectives, which have led us to the following design principles:

- *Choose a memory page as the unit of protection*: Memory is the prime resource in computing system. Many other resources are mapped to memory. The protection can be generalized if we can provide protection for memory. In addition, memory *page* is the fundamental unit of memory management. Many other operating system constructs, such as file systems, treat the page as their internal or external unit of resource abstraction. We can, therefore, achieve a simple and orthogonal interface by using a memory page as the unit of protection.

- *Protect information instead of resource*: As the target of protection, we choose the *information* stored in a page rather than the page itself, enhancing the generality and orthogonality of protection. This is achieved by encrypting the content of the page. By allowing an operating system full accesses (i.e., read/write/relocate) to the pages in their encrypted form, the operating system is not obstructed in managing memory and address space, yet the information stored in the pages is protected.

- *Avoid using operating system abstractions*: Operating system abstractions, such as process and address space, are artificial. At first, it seems easy and simple to use the process and address space as the principal and perimeter of the protection. However, relying on them imposes limitations unwittingly and actually hurts the simplicity and generality of protection. For instance, inside of the kernel, the notion of address space becomes imprecise. The process abstraction is temporally imprecise especially when it is created. Therefore, independent notions for protection principal and perimeter help enhance the generality and simplicity of protection.

2.4 SP^3 protection model

This section first provides an abstract definition of the SP^3 system in Subsection 2.4.1 and then illustrates with an example how it provides data privacy protection to user-level applications in Subsection 2.4.2. In Subsection 2.4.3, the application distribution and key management issues are presented. Subsection 2.4.4 discusses the security and safety of the presented SP^3 model.

2.4.1 Definition

As the principal of the SP^3 protection, we use the concept of protection *domain* [65]: access permission is determined based on the domain context. Each domain of a running SP^3 system is uniquely assigned and identified by an SID (*S*P^3 Domain *ID*) value. To identify the currently executing domain, the SP^3 system may keep a variable called *current* SID. The operating system is assigned an SID of 0. Therefore, the current SID is automatically switched to 0 when an interrupt or an exception occurs. In most cases, it is safe to consider a domain as a process, but a domain is not exactly the same as a process; multiple processes can share the same SP^3 domain. The kernel is always executed with an SID of 0.

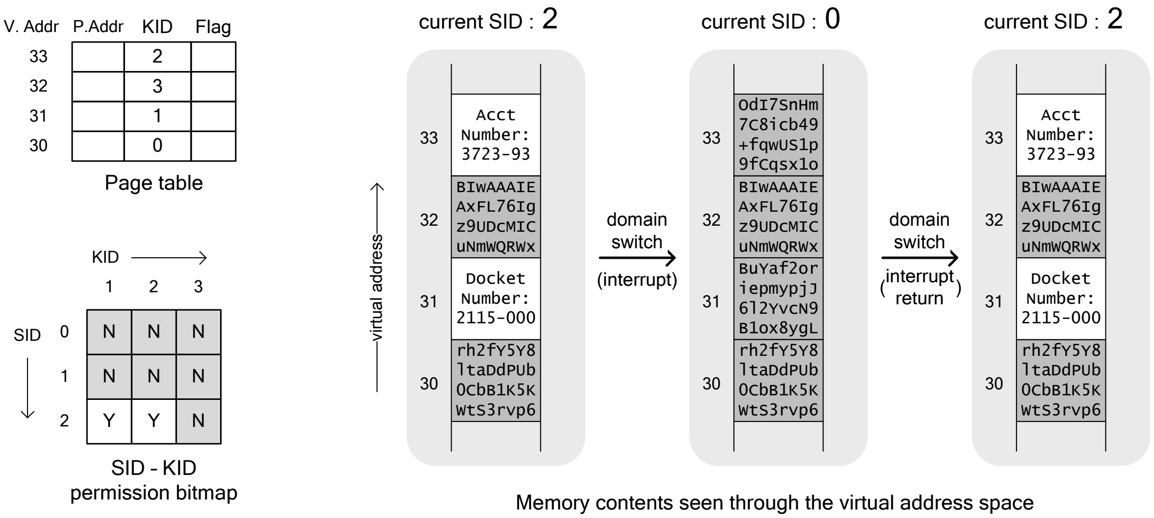
The definition of SP^3 is divided into three parts. First, the *secure paging* extends the interface of a general paging system to maintain the domain boundary. Second, the *secure domain switch* is responsible for safe domain crossing upon interrupts. Last, the *domain operations* handle the dynamics of domain creation and deletion as well as transferring access permissions for sharing. In this section, we only outline the SP^3 constituents, omitting details relevant to actual implementation, which will be addressed in Chapter III.

Secure paging: The Page Table Entry (PTE) structure of the machine's paging unit is extended to include a new multi-bit field, called KID (*K*ey *ID*), which is used to locate a symmetric key. The SP^3 system internally keeps a database that stores symmetric keys, called the *key database*. The KID value of a PTE serves as an index to the key database.

The SP^3 system also maintains a *permission bitmap* that tells which domain (identified by SID) can use which symmetric key (identified by KID). The operating system is prohibited from directly accessing the key database and the permission bitmap, but it is allowed to modify the KID field in a PTE. When a domain with SID s accesses memory, page tables are traversed for virtual-to-physical address translation. During the page traversal, the KID k of the matching PTE is checked against the permission bitmap to see if s can use k. If so, the SP^3 system renders the decrypted image of the physical page using the symmetric key indexed by k. Otherwise, the SP^3 system renders the verbatim image of the page. KID 0 is defined as the 'null' key, which always renders the verbatim image when it is used in a PTE. SID 0 is reserved for the domain of the operating system.

Secure domain switch: The current SID changes to 0 when interrupts or exceptions occur, since these events cause traps into the operating system. However, before the operating system takes over control, the execution context of an outgoing domain must be securely stored to prevent information leakage and hijacking of domain context. Thus, the value of the machine registers and the SID of the interrupted domain are encrypted, creating a *secure domain context*, which is passed to the operating system and then safely stored as an opaque data structure. The secure domain context is also tagged by an authentication hash to prevent overriding the SID.

Domain operations: For creation and deletion of domains, we define two operations, `Alloc` and `Free`. `Alloc` creates a domain by assigning an SID, loading symmetric keys to the key database, and initializing KID permissions by setting appropriate bits in the permission bitmap. Symmetric keys may be loaded via a key exchange protocol: a unique public key pair (K_P^+, K_P^-) is assigned to an SP^3 system. To deliver a symmetric key K_s to the system, $\{K_s\}_{K_P^+}$ is passed as an argument to `Alloc`, which uses K_P^- to extract K_s and store it in the key database. When a key is loaded, existing permissions are revoked for the KID to which the new key is indexed. `Free` deletes a domain by revoking the key permission



V. Addr	P.Addr	KID	Flag
33		2	
32		3	
31		1	
30		0	

Page table

SID \ KID	1	2	3
0	N	N	N
1	N	N	N
2	Y	Y	N

SID - KID permission bitmap

KID
SID
current SID : 2
current SID : 0
current SID : 2
virtual address
33
32
31
30
Acct Number: 3723-93
BIwAAAIE AxFL76Ig z9UDcMIC uNmWQRWx
Docket Number: 2115-000
rh2fY5Y8 ltaDdPUb OCbB1K5K WtS3rvp6
domain switch
(interrupt)
OdI7SnHm 7C8icb49 +fqwUS1p 9fCqsx1o
BuYaf2or iepmypjJ 6l2YvcN9 B1ox8ygL
domain switch
(interrupt return)
Memory contents seen through the virtual address space

Figure 2.1: An example SP^3 system showing a page table, permission bitmap, and memory contents. SP^3 renders different views of the virtual memory as the current SID changes. Domain 2 has permission to KID 1 and 2; thus it sees decrypted images at virtual addresses 31 and 33. Domain 0 – the kernel – has no permission to any keys; thus the memory contents are rendered as verbatim images. The current domain of the CPU switches from 2 to 0 upon interrupt, and back to 2 upon return from the interrupt.

and releasing the SID.

2.4.2 Examples

Figure 2.1 illustrates how SP^3 renders different views of the virtual memory as the current SID changes. In the figure, there are two active domains in the system. One is domain 2 created by the `Alloc` operation, which also loaded symmetric keys along the operations. The other domain is domain 0, which is predefined as the domain of the operating system kernel. In this example, the two domains share the same page table. The figure also shows a section of the page table with the KID values of PTEs. When domain 2 is executing, it sees decrypted images of the pages at virtual addresses 31 and 33. The two pages are decrypted by the symmetric keys referenced by KID 1 and 2. When an interrupt occurs, the current SID is changed to 0. In the center column of Figure 2.1, the operating system is running, but it cannot see the decrypted image at the virtual addresses 31 and 33, because it has

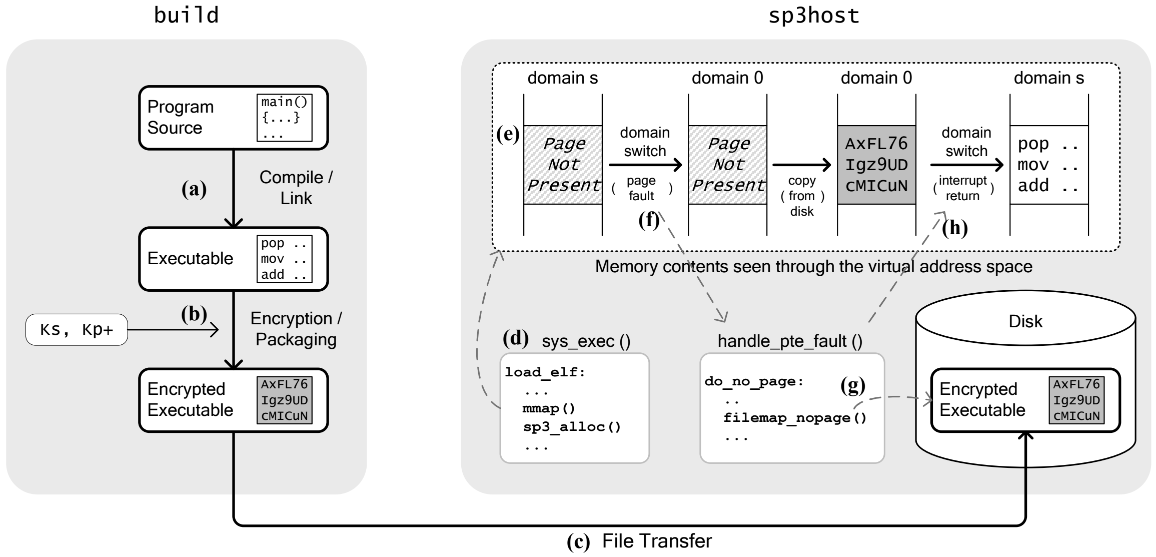


Figure 2.2: An example scenario of an application being generated, delivered, and executed on an SP^3-capable host computer. From a trusted computer `build`, an encrypted executable is generated (steps (a) and (b)) and then delivered (step (c)) to an SP^3 capable computer `sp3host` in which an untrusted operating system is running. Steps from (d) to (h) illustrate what happens when the application is being executed and how the encrypted executable image is ultimately viewed as a decrypted image by the application process.

permission to neither KID 1 nor 2. Instead, the operating system sees the pages' verbatim images, which are in their encrypted form. The current domain is switched again when the operating system schedules domain 2 and returns from the interrupt. The operating system uses the saved encrypted domain context for domain 2, which is executing after the domain switch, as seen in the third column. Domain 2 sees the decrypted images at the virtual addresses 31 and 33, according to the KID values of corresponding PTEs and permission bitmap entries.

Figure 2.2 depicts how a user application is generated from source code, transferred to an SP^3-capable untrusted system, loaded onto memory, and finally, executed on the system[2]. Shown on the left is a trusted computer `build`, where the encrypted executable is generated from a program source code. On the right, the computer `sp3host` is an SP^3 capable host computer on which an untrusted operating system is running. The untrusted

[2] Issues on this process are detailed in Chapter IV.

operating system is a modern commodity operating system such as Linux. The SP^3 layer of sp3host is assigned a public key pair $\{K_P^+, K_P^-\}$. K_P^+ is made known to build, but K_P^- is kept secret and only known to the SP^3 layer of sp3host.

In step (a) of the figure, an application source code is compiled on the host build in the same way other programs are compiled. In the program linking phase, it is ensured that the segments of the executable are aligned with page boundaries. Then, in step (b), the executable undergoes encryption using K_s, and the encrypted executable is packaged with a special header that contains $\{K_s\}_{K_P^+}$. This encrypted executable package is ready to be delivered to an SP^3 capable computer, which is sp3host in our scenario. In step (c), using a conventional file transfer method, the encrypted executable file is transferred to sp3host and stored on the disk. Because K_P^- is kept secret and K_s is protected, nothing involved in the transit of the file can steal the content of the executable. Now on sp3host, when a user executes the application, the executable is loaded via the exec() system call. As shown in step (d), the kernel's exec handler then calls on a binary loader, which overlays current virtual address space with the contents of the executable via memory file mapping such as mmap(). The binary loader also detects the special header and executes the Alloc operation. This Alloc creates domain s, loads K_s on KID k, and sets the permission bitmap on (s,k) (not shown in the figure). Due to the use of demand paging, the actual pages for the executable have not yet been allocated: the mmap() only reserves a range of virtual address space for the mapped executable file. Therefore, the pages for the executable and associated page table entries (PTEs) are marked as not-present, as shown in (e). The actual allocation of physical page frames and updating PTEs is to be handled via the page fault handler. When the application process starts to execute, it triggers first page-faults on these non-present pages (step (f)), invoking the page fault handler. Note that because of SP^3's secure domain switch rule, the current SP^3 domain changes from s to 0 when the page fault occurs. In step (g), the page fault handler loads the page by copying the executable image stored in the disk to a physical page frame. The page frame is then mapped into the virtual address by fixing the PTE that caused the fault. At this point, the operating system, running in domain 0, can only see the verbatim image of the program executable, which is the encrypted image of the actual program. When the PTE is being fixed, the KID field of the PTE is set to contain k. After this demand paging sequence, the application process resumes via return-from-interrupt in step (h), which changes the current domain from 0 to s. The resumed application process now sees the decrypted images in its virtual address space that caused the initial non-present page fault. From this point, running within the context of the SP^3 protection domain s, the loaded application sees the decrypted contents of the memory pages from its virtual address space.

Although this program loading example looks quite complicated, an application generally does not have to be concerned about this process. This is because the details involving the demand paging scheme are hidden and done transparently from the perspective of the application. Also, accessing the decrypted data in memory requires minimal effort from the application: all the application requires is to have the operating system map the pages with appropriate KIDs. The application does not have to call special functions nor does it have to set up special barriers in its code. The SP^3 has a flexible interface that allows the application to set up different cryptographic keys to different virtual addresses. This is achieved easily by using different KIDs. Also, for the memory regions that are not to be encrypted, the application can just use a null (0) KID.

2.4.3 Key management

The public key pair $\{K_P^+, K_P^-\}$ assigned to the SP^3 layer plays a central role in ensuring secure delivery of symmetric cryptography keys that are being used to encrypt application programs. In this case, the public key cryptography implements the key exchange protocol: the symmetric keys are encrypted by the public key K_P^+ and then safely delivered to the SP^3 layer, which can retrieve the symmetric keys by decrypting them using the private key K_P^-.

As with any public key cryptography, the public key K_P^+ is made public, meaning the public key is known to the program vendor, the end user of the program, and the operating system. It is safe to do so as long as the private key K_P^- is kept secret. The private key is embedded in the underlying SP^3 layer and protected from the untrusted operating system. The private key is never known to the program vendor, the end user of the program, nor the operating system.

Symmetric keys to be delivered to a target SP^3 system are selected by the application vendor. The application vendor determines the value of the symmetric keys, and SP^3 does not impose any rules on the key selection. For example, the vendor can pick a unique key value for each application instance. Or, the vendor can use the same key for encrypting a set of application instances. If the program data is shared with another trusted party that is using its own symmetric key, the vendor might use the key for the shared data. Which key selection method to use is up to the discretion of the application vendor.

The application vendor encrypts the chosen symmetric keys with the public key K_P^+ of the target SP^3 system. The symmetric keys can then only be retrieved by the target system's underlying SP^3 layer. The symmetric keys are safely stored in the SP^3 layer, which is more privileged than the operating system. This process achieves the goal of delivering the symmetric keys securely from the vendor to the target SP^3 layer without trusting end users or any part of the operating system. The `Alloc` operation is the interface that triggers the underlying SP^3 to decrypt and load the keys.

The `Alloc` operation is atomic from the perspective of the software that initiates the operation. It results in two possible outcomes: pass or fail. If `Alloc` passes, a new SP^3 domain is created, the symmetric keys are decrypted and loaded into the key database, and the corresponding permission bits are set or reset. The operation then returns to the caller the new SID number, the KID numbers and a secure domain context that serves as the initial entry into the program. If `Alloc` fails, the state of the underlying SP^3 system does not change. `Alloc` fails when the input is invalid or there are not enough resources to handle the request.

In the real world, the public key pair assigned to the SP^3 system should be generated by a trusted third party such as a Certificate Authority (CA). The CA authenticates and signs the public key so that the vendors can verify the legitimacy and authenticity of the public key. Otherwise, someone can generate a public key/private key pair and submit the public key to the application vendor in an attempt to retrieve the symmetric keys as well as the application contents. Further issues regarding key distribution and management, such as key revocation or key escrow, are important but largely out of the scope of this thesis.

The question of whether end-users are adversary has multiple answers depending on the context. For example, when we are using SP^3 to protect a media player program from reverse-engineering by an end-user, we can say that the end-user is the adversary. In this case, we use SP^3 to grant the end-user only the right to execute the program and no more. On the other hand, when we are using SP^3 to protect, for instance, an ssh client program from being spied on by a malicious system administrator, the end-user of the ssh program is not the adversary but the one we want to protect. In the case of donation-based distributed computing where we use SP^3 to secure the execution of the distributed worker programs running in the untrusted remote hosts, there is no such end-user that interacts with the program in the remote hosts. Instead, the end-user in this case can be more correctly defined as the person who initiates and controls the whole distributed computation.

2.4.4 Security of SP^3

Due to the use of the public key cryptography and the `Alloc` interface, the symmetric keys and the application contents encrypted by the keys are never revealed to the operating system, or any entities involved in the transit of the encrypted application. It should be emphasized that the program loader of the operating system does not have to be trusted. The program loader invoked as a part of `exec()` is usually the place where the encrypted application binary is mapped to the address space and the `Alloc` operation is called. However, the program loader is in no way allowed to obtain the decryption key or set the permission bitmap arbitrarily. Note that this does not prevent the loader from performing the program loading functions: to map the binary, the loader just treats the encrypted images as-is and then sets the KID field of the PTEs accordingly.

An attempt to somehow deceive the system by incorrectly setting the KID field of the PTEs cannot succeed because the permission bits are not set for the SID-KID combinations other than the ones created by the `Alloc` operation. If the KID field is set incorrectly, the page will be rendered encrypted because the permission bit is not set. Also prevented is the scenario of creating the same SID-KID permission bitmap for two different programs by repeatedly creating and deleting SP^3 domains and using this to somehow compromise a trusted program, because the corresponding KID entry in the key database will contain different symmetric keys.

In addition, although the externally visible SID number can be small and possibly recycled if many SP^3 domains are created and deleted, SP^3 internally uses extended and hashed SID numbers to prevent overriding the SID number, as detailed in Section 3.4. This prevents potential abuses of reusing secure domain contexts saved from a previous SP^3 domain instance with the same SID.

2.5 Implementation alternatives

The system that implements SP^3 forms a trust base whose execution must be more privileged than the operating system. In today's computing environment, SP^3 can be realized in three different ways: direct hardware modification, using a hypervisor (software), and a software/hardware hybrid. Next we present these alternatives and discuss their advantages and disadvantages.

2.5.1 Modifying hardware

One can realize SP^3 by directly modifying the hardware. Obviously, the hardware itself is more 'privileged' than the operating system, and thus, provides a safe and secure implementation base for SP^3. The implementation of the extended PTE field, access control logic, and domain operations is straightforward. For the actual encryption of memory pages, one may utilize encryption hardware that can perform a two-way encryption on the memory cache boundary [67, 98]: to render the content of a decrypted page, the encryption hardware is activated to fill the corresponding cache line with the decrypted memory content.

The greatest advantage of this hardware-based implementation is its superior performance: hardware can perform much faster cryptographic operations than software. Hardware can also directly support the extended paging interface. Another advantage of this is that hardware can provide a more secure trust base than the software approach.

However, this approach has the biggest practical disadvantage: it cannot be deployed to existing systems due to its requirement of hardware (especially processor) modifications. Another disadvantage of this is that specific machine details become highly relevant to the viability of the actual construction of hardware. For instance, encryption based on the cache line works only in a uniprocessor system.

2.5.2 Using a hypervisor

SP^3 can be implemented using a hypervisor, also known as a virtual machine monitor (VMM). A hypervisor, positioned between hardware and operating system, is system software that can create multiple virtualized hardware instances to be multiplexed upon a single physical machine, making it possible to run multiple operating systems concurrently. A hypervisor is also used to realize hardware extensions without actually changing the real machine, or to implement system services below the operating system. For its own protection, a hypervisor runs with more privilege than the operating system and has a safe perimeter. Therefore, SP^3 can be safely implemented using a hypervisor. Chapter III details a SP^3 system that we built by modifying a full-fledged hypervisor.

Using a hypervisor has the following advantages. A hypervisor-based SP^3 system is readily deployable in existing systems as it does not require any hardware modification. Compared to the hardware-based approach, it does not suffer from such machine specifics as multiprocessing or cache consistency issues, since the hypervisor abstracts away such details. The hypervisor can easily support multiprocessor or multi-core systems by sharing the permission bitmap and key database among the processors and by having each processor have its own current SID variable.

The biggest disadvantage of this approach is its poorer performance compared to the hardware-based counterpart, since software has to perform encryption and emulate the extended paging system. Another disadvantage is that the size of the trust base may not be optimal because the hypervisor usually comes with other features. However, we do not need the ability to run multiple operating systems since a hypervisor is used only for the purpose of implementing a service below the operating system. Therefore, we may reduce the size of the trust base by implementing an SP^3 system only with the virtualization techniques required to achieve a safe perimeter.

2.5.3 Using a hardware/software hybrid

A hardware/software hybrid solution can exploit the advantages of both software- and hardware-based approaches. There are many possible ways to decide which part should be implemented in hardware, but we do not discuss this further since it is out of the scope of this thesis. The disadvantage of this approach is that it still requires hardware modifications, albeit to a much lesser extent than the pure hardware approach.

2.6 Discussion

Many SP^3 design choices might not seem too obvious when SP^3 is introduced abstractly. For example, one might wonder why SP^3 allows the operating system to access encrypted memory views; can it be much simpler if we just prevent the operating system from accessing decrypted views, thus eliminating the whole encryption and decryption? Or, why should each SP^3 domain be associated with multiple symmetric keys? If we had just used a single key for each domain, the KID field would not be necessary. In this section we discuss and justify the choices we made in the design of SP^3 by answering the following questions.

- Why do we let the operating system see the encrypted views?
- Why should each SP^3 domain be associated with multiple symmetric keys?
- Why does the page table entry have to be modified to include a KID field?
- Why should the SP^3 domain be an independent notion separate from the concept of process?

2.6.1 Encrypted view for the operating system

The reasons why we render encrypted views to the operating system are threefold. First, operating systems do need to access user-space memory in order to perform memory management. For example, upon memory pressure, operating systems can initiate disk swap, and the swapper process has to access the contents of evicted pages. In most cases, the actual disk transfer is done by a disk controller performing Direct Memory Access (DMA), so an operating system does not technically have to read the memory content itself, but the disk controller can always fall back to the Programmable I/O (PIO) mode of operation where the operating system has to read memory content. In addition to swapping, memory management algorithms can relocate the physical pages at runtime in order to increase performance or to save power [52]. The second reason is that we want application programs to intentionally exploit the fact that the operating system can only see encrypted images. For example, applications can save encrypted files using memory file maps. The third reason is that the alternative — a solution that does not even render encrypted views — can be more intrusive to operating systems because we need to physically block any access attempt made from operating systems.

2.6.2 Multiple keys per domain

It is very useful to assign multiple symmetric keys to each SP^3 domain. For example, if an application needs to encrypt different files or memory buffers with different keys, SP^3 offers a very easy method for doing the encryptions. Later, in Chapter V, we implement a secure media player that exploits this capability.

2.6.3 Extension on the page table entry

Since we allow multiple symmetric keys to be used in a single protection domain, we have to find a way to specify which key to use for the memory regions of the protection domain. We find that augmenting the page table entry structure is effective, comprehensive, far-reaching, and less intrusive a way of specifying that information. For instance, by introducing KID in the page table structure, we can orthogonally extend the kernel's memory management routines and the `mmap()` interface. An alternative approach would be a solution based on segmentation, which would be very difficult to deal with.

2.6.4 Protection domain separate from process

There are two reasons why we use an independent notion for protection domain. The first reason, which is less important, is that we can share the protection domain for multiple processes of the same application, thereby saving resources. The second reason, which is more important, is that it is actually very difficult to use the concept of process as the security principal. Let us imagine what it would be like if we are using process as the principal. Right from the beginning, we have a problem dealing with interrupt because interrupt causes a transition to the untrusted operating system but the currently running process is unchanged; the process has just started to execute in the kernel context. Let us further imagine that we have solved this problem by introducing additional modes for processes running in the kernel context. Then, a complexity arises when the kernel performs a process context switch. Since we are relying on the process for security principal, we have to change the underlying security context upon the process switch. But exactly when do we change the underlying security context? Upon switching page table base register? Upon replacing the kernel stack? Upon reloading the register file? Or some other place? Even if we decide the exact point when the underlying security context changes, we have to verify the decision by making sure there is no 'loophole' that can be exploited. All these issues can be avoided if we use an independent notion for protection domain, as we do in SP^3.

2.7 Conclusion

We conclude this chapter by highlighting the quality and soundness of SP^3 with respect to our design objectives. The observations and claims given in this qualitative analysis are backed up by an actual implementation, which comprises the remaining chapters of this thesis.

First, with a degree of elegance, SP^3 resolves conflicts between the need for managerial privileges and the restrictions mandated by the protection system. With SP^3, the operating system can retain virtually all power of managing processes and memory. The only exception is that it cannot obtain meaningful information from user memory, which is fine most of the time because the operating system is usually indifferent to the content of user memory. But there are issues with some operating system constructs because they were implemented to directly refer to user memory. For example, using a user stack for passing system-call parameters or saving processes' contexts upon signals violates the rules of SP^3.

Therefore, changes need to be made to these constructs (Chapter IV). However, compared to the scale of the protection provided by SP^3 , these changes are relatively minor.

Second, SP^3 defines a simple and narrow interface, thus minimally affecting existing software. For operating systems, adapting to the KID field only requires an orthogonal extension to the memory-management routines. No compiler modification is needed to create SP^3-enabled applications. A standard set of binary utilities and a few compiler tricks suffice for generating application executables. The protection is provided in terms of the data's *position* and the process's *context*. For programmers, providing data privacy in their program is simple and easy since they only have to place sensitive data in the protected memory.

Third, SP^3 provides far-reaching protection, without much restriction. For example, applications can freely apply SP^3 protection to any region of memory in its virtual address space. At the same time, on-demand paging, memory-mapped files, disk swapping and memory sharing work equally on the protected memory. Using memory-mapped files, SP^3 enables applications to write files to disk in encrypted form without trusting/requiring a special secure file system.

Last, SP^3 achieves a high degree of implementation freedom.The interface that SP^3 defines is platform-independent: changing the hardware in one platform to implement SP^3 would not be particularly easy or hard because of CPU flavor. In case the target system is virtualizable, SP^3 can be implemented without modifying the hardware, as detailed in the next chapter.

CHAPTER III

HYPERVISOR-BASED SECRECY PROTECTION

3.1 Introduction

Hypervisor, a virtual machine monitor that directly runs on bare hardware, is becoming popular and has already penetrated deeply into modern computing environments [13, 107]. Hypervisor is not only attractive in consolidating servers and planning server resources, but also advantageous in enhancing system security.

Hypervisor can provide a perfect implementation point for many security applications because it can be inserted between hardware and operating system. For instance, many intrusion-detection systems based on hypervisors have been proposed [44, 59, 62]. Hypervisors have also been utilized for providing security services to upper-layer software. For example, hypervisors can provide virtual instances of Trusted Platform Module (TPM) of the trusted computing architecture [16, 43]. Hypervisors used for enhancing security rely on the property that they form a relatively small and easy-to-secure trusted computing base.

In this chapter, we propose a novel usage of the hypervisor for implementing a new layer of protection. This protection layer directly secures the memory contents of user-level applications, guaranteeing protection even from a malicious or faulty operating system. This protection is achieved by encrypting the contents of the user memory pages; when a program accesses a memory page, the hypervisor determines which image of the page to provide to the program. Whether to use a decrypted image of the page or a verbatim (hence encrypted) image is determined by the access permission of the program accessing the page.

Our protection system results in a very powerful secrecy protection infrastructure that can secure the entire execution of a user-level application. The sensitive information of a user application, including both code and data, is guaranteed to be protected against malicious or faulty operating systems. Therefore, unless the hypervisor itself is compromised, the secrecy of an application's memory contents and relevant execution context is preserved even when the operating system has been compromised.

The semantics and interface of this page-based encryption system are abstractly defined in the previous chapter: the *Software-Privacy Preserving Platform* (SP^3) is the model that our hypervisor-based protection system is implementing. In SP^3, a protection *domain* is defined as the principal in which a set of access permissions associates domains with a set of cryptographic keys. These access rights govern the ability to use the keys, which are to encrypt/decrypt memory pages. If an application program is running inside an SP^3 domain, the application sees the decrypted memory contents through the virtual address space; programs outside the domain, including the operating system, may only see the encrypted memory contents.

In this chapter, we describe modifications and extensions made to the hypervisor to implement SP^3. To encrypt pages and secure the SP^3 domain boundary, SP^3 extends the semantics of the paging system and interrupt interface of a CPU. We make the hypervisor emulate the extended paging and interrupt semantics. We also discuss and evaluate techniques to improve the hypervisor's emulation performance.

Page-frame replication is a way to reduce the overhead of encryption. In this technique, a hypervisor retains page frame copies containing decrypted images of an original page. When a decrypted image of a page needs to be supplied to a program, the hypervisor manipulates the page table entry (PTE) to redirect requests to the page frame copy containing the decrypted image. This reduces the number of costly cryptographic operations that would have to be performed on the entire page frame.

Lazy synchronization is also used to further reduce the number of page-wide encryptions. Under page-frame replication, synchronization is needed when an update occurs to one of the replicated images. This synchronization propagates the update to the other images by page-wide encryptions. However, with lazy synchronization, this costly synchronization is deferred as long as possible: the synchronization happens not when an image is modified, but when one of the other images is accessed. This technique turns out to be very effective because no synchronization takes place unless two SP^3 domains actively access the two related images simultaneously.

We modified the Xen hypervisor and Linux kernel to implement a hypervisor-based SP^3 system. The modified Xen, serving as the trusted computing base for user-application secrecy protection, implements the full semantics of the SP^3 model. Linux, running on top of the modified Xen, is thus removed from the trust base of user applications that are running within SP^3 domains. If Linux violates the protection rules, it will at worst crash the system, but protection of the applications' memory secrecy is guaranteed.

We evaluated the Xen-based implementation by measuring the runtime performance of SP^3 applications. Our evaluation results indicate that applications running in the modified Xen experience 0-23% slowdown compared to the same applications running in the native Xen environment. The result also confirms the efficacy of page-frame replication and lazy synchronization schemes.

The rest of this chapter is organized as follows. Section 3.2 provides background on Xen and Linux's memory management and interrupt handling. Section 3.3 presents the key ideas for realizing SP^3 protection in a hypervisor. Section 3.4 details our SP^3 implementation on Xen. Section 3.5 evaluates the implementation. Section 3.6 discusses related work.

3.2 Background

This section provides background on the internal workings of hypervisor and operating systems. This background information is intended to help the reader understand the description of our hypervisor-based implementation of SP^3, which will be presented in Section 3.4. In this section, we particularly focus on paging (Section 3.2.1) and interrupt interface (Section 3.2.2), as they are closely related to the description of our implementation. We summarize with an example (Section 3.2.3) by walking through what happens when we execute a program in a virtual machine environment.

In what follows, we assume Linux running on the x86 [54] architecture as the choice of computing platform. We also use Xen as our choice of hypervisor. Despite this choice of specific platforms, our discussion below can be equally applied to many other general processor architectures, operating systems and hypervisors.

3.2.1 Paging

Paging is the fundamental facility for memory management in contemporary systems. Supported by a hardware Memory Management Unit (MMU), a physical memory page is mapped to a virtual address space via the page table entry (PTE) structure. The MMU translates a virtual address to a physical address by page-table lookup using the virtual address to find a PTE that contains the physical page frame number. Each PTE also contains bit flags such as Present (`P`) bit (accessing a page with `P` bit cleared causes a non-present page-fault), Writable (`W`) bit (writing a page with `W` bit cleared causes a read-only access-violation page-fault), and Dirty (`D`) bit (the processor sets `D` bit when data has been written to the mapped page).

Without a hypervisor, operating systems directly manipulate the MMU data structure to implement the virtual address space and various paging tricks, such as demand-paging, copy-on-write, virtual memory, and disk buffer cache.

With a hypervisor present, an operating system runs on a virtualized hardware platform where the operating system is given a “physical memory” of virtual machine that is an illusion created by the hypervisor. Running between the bare hardware and operating systems, the hypervisor adds another layer of address translation. One way to implement this translation layer is to use shadow page tables [107]. In this technique, a guest operating system’s page tables are “shadowed” by real page tables to be directly used by the processor. The hypervisor intercepts all references and updates to the guest operating systems’ page tables, performing additional translation, which is called “physical-to-machine” translation.

In para-virtualized systems where operating systems are modified to run on a virtual machine, part of the “physical-to-machine” translation is performed by the guest operating system. This is to avoid complexity and overhead that would otherwise be incurred in a fully-virtualized system. Although a para-virtualized system directly exposes MMU states to the guest operating system, the hypervisor still enforces strict rules regarding MMU and page table updates, thus guaranteeing safety to the hypervisor.

3.2.2 Interrupt

If the processor receives a hardware interrupt or generates an exception, it suspends its execution of the current program in order to serve the interrupt or exception. The processor saves the context of the interrupted program for later use when the program is to be resumed. In the x86 architecture, this context, called “exception frame,” is saved in the kernel-mode stack upon interrupt. The interrupt is usually the point where the kernel is entered; it causes the processor to vector to the kernel’s interrupt/exception handler and the processor mode is switched from user mode to privileged mode.

Without the hypervisor, operating systems directly handle interrupts. An operating system directly programs interrupt vector tables to cause the processor to jump to the appropriate handler code in the kernel. The handler then performs appropriate actions to handle the source of the interrupt or exception. Upon completion of handling the interrupt event, the kernel runs a scheduler to select the next program to run. To switch the context to the selected program, the kernel executes an instruction called "return from interrupt" with the saved exception frame as the argument of this instruction. The processor switches back to the user mode and resumes execution of the user program.

With the hypervisor present, however, the hypervisor intercepts every interrupt and exception. It examines the cause and nature of the interrupt and then decides whether to handle the interrupt itself or to forward the interrupt to the guest operating system. When it decides to forward the interrupt, it creates an exception frame on the guest operating system's kernel mode stack to emulate the processor's behavior. The content of this exception frame can be programmed by the hypervisor to suit its needs.

From an operating system's perspective, the underlying hypervisor's involvement is completely hidden in the case of full virtualization. In the case of para-virtualization like Xen, the operating system is required to be modified to use the para-virtualized interrupt interface. Nevertheless, the para-virtualizing hypervisor is able to intercept every interrupt and exception, and thus is fully protected from guest operating systems.

3.2.3 Example

Here we summarize paging and interrupt in a virtualized environment by using an example where we walk through an application being executed. When a user application program is first executed by the `exec` system call, the kernel routine handling `exec` loads the binary (e.g., ELF executable) to read the program header information. Then, the kernel maps code, data, and stack area to the process address space. At this time, the operating system only assigns virtual memory regions and memory is not assigned; the corresponding PTEs for the regions are cleared with their `P` bit. This is because of the demand-paging scheme. The actual mapping occurs when non-present page-faults on these unmapped pages are handled.

During these events of system call, PTE manipulation, and page-faults, the hypervisor intervenes to virtualize hardware by page-table shadowing and forwarding interrupts: each non-present page-fault vectors first to the hypervisor's handler. If the hypervisor determines the fault should be handled by the guest operating system, it forwards the fault to the guest operating system. Then the page-fault handler in the guest operating system allocates and maps a physical page to the faulting address by updating the corresponding PTE. This PTE update is also intercepted by (or submitted to) the hypervisor for its implementation of shadow paging.

3.3 Realizing SP^3 model in hypervisor

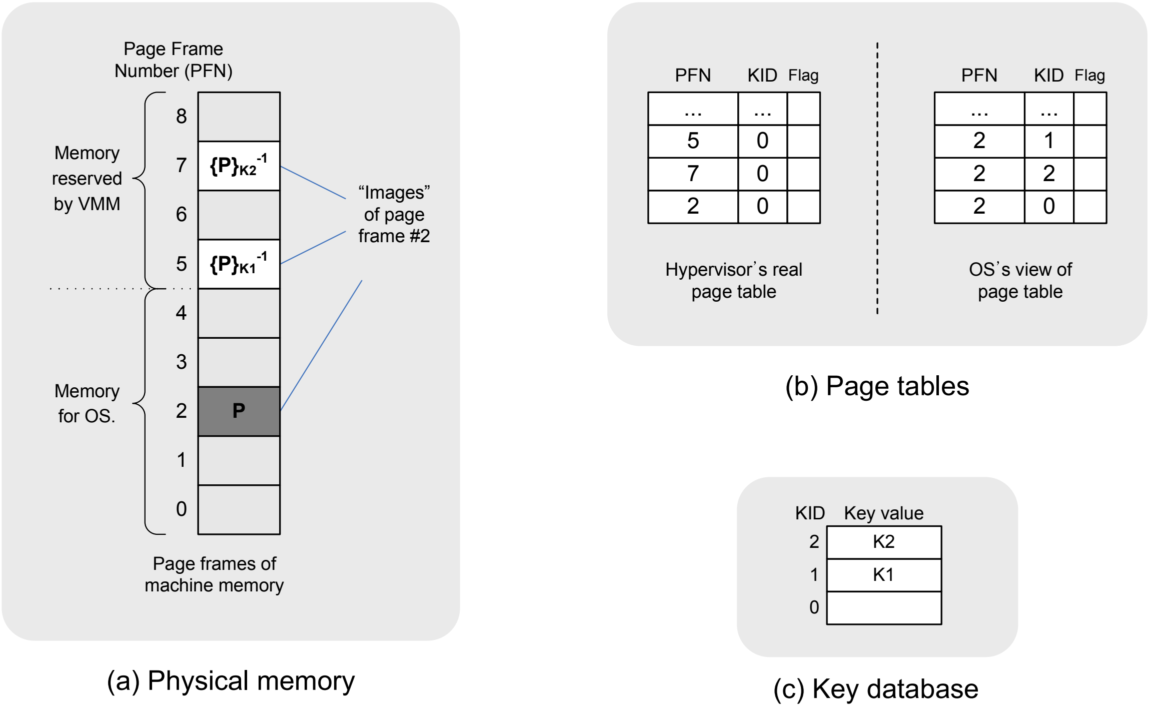
In this section, we describe how to realize the SP^3 protection model using a hypervisor. We first present how to efficiently emulate SP^3 secure paging by introducing page-frame replication and lazy synchronization. Then we discuss how to realize SP^3 secure domain switch by changing interrupt semantics. Finally, we provide information on how to emulate SP^3 domain operations.

3.3.1 SP^3 secure paging

As defined in Section 2.4.1, the heart of the SP^3 system is the SP^3 secure paging, which is capable of rendering different views of the same page frame. That is, the page frame referenced by a PTE with non-zero KID should be rendered as decrypted if the page is accessed when the current SID has the permission to use the KID. We now discuss how to use a hypervisor to emulate such semantics of SP^3 paging.

In the design of hypervisor-based emulation of SP^3 paging, we should consider the performance impact of encryption. To provide a decrypted view of a page, the hypervisor should perform software decryption on the page, the size of which is typically 4KiB. Obviously, a naive design would incur significant run-time performance overhead. Thus, we would like to minimize the performance overhead by using two schemes that can minimize the number of cryptographic operations as described below.

Page-frame replication is the primary vehicle for efficient emulation of SP^3 paging. In



(a) Physical memory

Hypervisor's real page table

PFN	KID	Flag
...	...	
5	0	
7	0	
2	0	

OS's view of page table

PFN	KID	Flag
...	...	
2	1	
2	2	
2	0	

(b) Page tables

KID	Key value
2	K2
1	K1
0	

(c) Key database

Figure 3.1: Hypervisor implements SP^3 paging by page-frame replication. Shown in (a), the hypervisor keeps decrypted copies of an original page frame (PFN 2) in different memory locations (PFN 5 and 7). The hypervisor uses one of these page frames when the original frame is mapped with a PTE with a KID value. The redirection of page frame is performed transparently by manipulating page tables as shown in (b). Note that the KID field does not contain actual key, but it contains a index to the symmetric key database shown in (c).

this scheme, the hypervisor maintains copies of decrypted images of a page frame. Each of the decrypted images contains the decryption result on the original page using a particular symmetric key. The hypervisor keeps these images in its privately-maintained memory area. Rendering a decrypted view of a page is thus realized by redirecting the page to one of the decrypted images. The hypervisor can realize this redirection by virtualizing access to the page tables; it intercepts modifications on page tables to realize the extended KID field, and it induces page-faults to provide the hypervisor the opportunities to check the permission and to perform actual redirection. These operations are directly handled by the hypervisor and thus hidden to the operating system.

Figure 3.1 illustrates how a hypervisor implements the page-frame replication scheme.

In Figure 3.1(a), physical page frame number (PFN) 2 has two decrypted images located on PFN 5 and 7, each of which is the decryption of PFN 2 using the symmetric key selected by KID 1 and 2, respectively.[1] Note that the actual symmetric keys (K1 and K2) are selected from the key database (Figure 3.1(c)), which resides inside of the hypervisor. Figure 3.1(b) shows page tables virtualized by the hypervisor. On the right side is the virtualized page table, which the operating system can modify. The virtualized page table is shadowed by the real page table, which the MMU refers to. In the figure, the operating system programmed the virtualized page table such that PFN 2 is mapped in three different PTEs with different KID values of 0, 1, and 2. The corresponding PTEs in the real page table then contain PFN 2, 5, and 7, and thus, the hypervisor renders decrypted views on the same page, realizing the SP^3 paging semantic.

Although keeping decrypted images reduces the number of cryptographic operations, those images must be synchronized if one of the images or the original page gets modified. The synchronization is necessary for providing consistent views on all images; if a program modifies a decrypted image, then the original page, although its content is encrypted, must reflect the change when accessed later. Obviously, this involves cryptographic operations and, unless properly handled, incurs high runtime overhead.

We solve this problem by employing *lazy synchronization*, which reduces the number of synchronizations among the images by delaying update propagation until the last minute. Synchronization is performed only to the pages that need to be updated and only when it is necessary; the synchronization happens not when one image is modified, but when one of the other images is accessed. This is realized by keeping track of the most-recently updated image among the images, including the original. Tracking the most-recently updated image is achieved by checking `D` (dirty) bit of PTE. The content of the most up-to-date copy is propagated to one of the 'stale' pages by means of the hypervisor's page-fault handler.

[1] Actual hypervisor adds another layer of address indirection by which a "physical address" (the virtualized address local to a virtual machine) translates to a "machine address" (the physical address of the underlying hardware). To simplify the discussion, we ignore this translation layer.

The hypervisor clears P bit of those stale pages to induce a page-fault through which the hypervisor can propagate updates behind the scenes.

The lazy synchronization scheme is highly effective because it exploits the fact that there are limited occurrences of active sharing in application programs: if a page is not accessed during the activation of the particular SP^3 domain, it will not generate any page-fault. Therefore, the images of a page frame are synchronized only when necessary, thereby reducing the runtime overhead of re-encryption for synchronization. Note that this lazy synchronization does not incur any encryption overhead for most of the normal application execution scenarios because page frames are not shared among different SP^3 domains.

3.3.2 SP^3 secure domain switch

The SP^3 secure domain switch extends the interface of interrupt and exception. To recap, the current domain switches to the operating system's domain, SID 0, when an interrupt or exception occurs. Also, upon occurrence of these events, the execution context of the outgoing domain must be securely stored in the 'secure domain context' to prevent information leakage and hijacking of the domain context. We now discuss how to emulate such SP^3 interrupt semantics in a hypervisor.

We can realize the transition of current domain by intercepting every interrupt and exception generated by hardware. Hypervisors are, by definition, capable of intercepting all interrupts and exceptions. When the hypervisor forwards an interrupt to a guest operating system, the hypervisor can change the current domain by setting current SID variable to 0.

The secure domain context, which contains register contexts and SID of the outgoing domain, is realized by extending the exception frame structure. As briefly described in Section 3.2.2, the processor generates an exception frame into the kernel mode stack upon an interrupt, and the hypervisor already simulates this behavior to virtualize interrupts. We extend this exception frame to contain a secure domain context. Thus, this extended exception frame has a new field for general-purpose registers (GPRs) and for SID value of the outgoing domain. These fields are encrypted and hashed. When the hypervisor forwards an interrupt to a guest operating system, it generates this extended exception frame instead of the original one.

The GPRs are cleared when the hypervisor raises a virtual interrupt by generating a secure exception frame. Upon receipt of this interrupt, the guest operating system will find the GPRs to be zeroed out. This is to prevent information leakage upon domain switch, because the operating system is untrusted.

After handling the virtual interrupt, the guest operating system requests the hypervisor to perform a 'return-from-interrupt' operation using the extended exception frame that has been saved from a previous interrupt. Upon receipt of this request, the hypervisor processes the extended exception frame to restore GPRs and SID value.

3.3.3 SP^3 domain operations

In the hypervisor-based realization, the domain operations are basically requests made to the hypervisor. Therefore, the interface for the domain operations could be simply realized by creating a new hypercall entry for each domain operation. However, we can alternatively achieve this by creating virtual 'instructions' for the domain operations. Execution of this instruction opcode will generate an 'invalid-opcode' fault, which should be captured by the hypervisor. The hypervisor will then examine the opcode to perform the matching SP^3 domain operation.

We favor defining new instruction opcodes rather than extending hypercall entries, because by creating new opcodes, the entire SP^3 interface looks as if the processor were supporting SP^3: from the perspective of an operating system, there is no functional difference between the hypervisor-based implementation and direct-hardware modification. After all, using the 'invalid-opcode' fault has no performance disadvantage over extending hypercall, because a hypercall is also implemented by generating a software interrupt.

3.4 Implementation

We modified the Xen hypervisor [13], which runs on top of x86 (IA-32) architecture [54]. Xen runs with higher privilege than the virtual machines it manages, and thus, it has a safe perimeter against operating systems.[2] Note that Xen's administrative virtual machine, known as dom0, cannot access the private area of Xen, therefore guaranteeing safety.

One unfortunate name collision needs to be resolved before we proceed. In Xen-terminology, a "domain" refers to a virtual machine instance created by Xen. In this thesis, this usage is avoided to eliminate confusion with our SP^3 protection domain. Henceforth, Xen's "domain" is referred to as 'virtual machine', and we use 'SP^3 domain' or simply 'domain' to refer to the SP^3 domain.

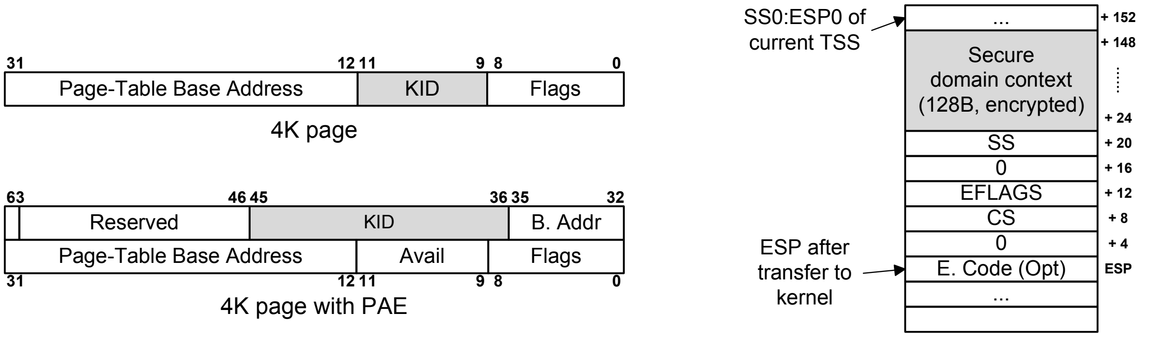
In this section, we first describe the implementation of emulating the modified interface of extended x86 architecture for SP^3 support. Then, we detail the realization of our design on the hypervisor, focusing on the mechanisms to efficiently emulate the SP^3 secure paging.

3.4.1 Implementing modified x86 interface

It is straightforward to incorporate into Xen the data structures directly related to the SP^3 protection model. We modified Xen to keep variables for storing the permission bitmap and cryptographic keys. To identify which SP^3 domain is executing in the system, an integer variable called `current_sid` is created to store the SID value of the currently executing SP^3 domain.

It becomes more complicated when we make Xen emulate the new extensions to the CPU-level interface, specified in Figure 3.2. The extensions reflect the new KID field in PTE structure, and generate a secure interrupt frame upon interrupt. Obviously, we did not actually modify the hardware; the specification given here is used as the reference interface

[2] We can say that Xen and its SP^3 extension, implemented within, can form a small and secure trust base, provided Xen is securely bootstrapped and attested. Secure bootstrapping and providing integrity measures are important for securing a trusted computing base, but description of this process is out of the scope of this thesis. We refer the readers to secure bootstrapping [60,71] for safe and secure loading of the Xen hypervisor.



(a) Extended x86 page table entry

(b) Modified x86 exception frame

Figure 3.2: Extensions made to x86 page table entry (PTE) and exception frame structures. The extended PTEs include a new multi-bit field for KID value. The secure exception frame, which is generated on the kernel stack upon interrupt, is larger than the original exception frame to contain fields for the GPRs and SID of the outgoing domain.

that Xen ultimately emulates.

Figure 3.2(a) shows the modified PTE structures into which the KID field is integrated. In its 'native' paging mode, the original x86 has 3 bits available for the KID field.[3] In its Physical Address Extension (PAE) paging mode, which has an expanded PTE structure, 27 bits are available for the KID field. The actual number of bits required for the KID depends on the size of the required KID space. For instance, when 10 bits from the PAE-enabled PTE structure are selected as the KID field, as shown in the figure, it allows the KID space to range from 0 to 1023.

We modified Xen to emulate this PTE extension by adding a code that can interpret the KID field. This code is added to the Xen's handler routine responsible for PTE updates. This handler routine is always invoked when a guest operating system modifies a PTE to map a page. Since MMU updates are sensitive, Xen makes sure it intercepts all PTE updates. In the para-virtualized environment of Xen, operating systems can update a PTE either by making a PTE-update hypercall, or by directly modifying the PTE. Either

[3] In fact, these bits are intended to be utilized by the operating system. But Linux, the operating system we use, does not utilize them.

way, Xen can always intercept the PTE update: a hypercall causes trap to Xen by definition; a modification to a PTE incurs a page-fault since the pages used as guest page tables are always mapped with W bit cleared, meaning any attempt to write to the guest page tables causes an access-violation page-fault, trapping into Xen. Therefore, by modifying the Xen's handler for PTE updates, the safe and transparent implementation of the extended KID field can be achieved. A guest operating system can update a PTE as if the hardware supported the KID extension.

Another modification we made to CPU-level interface is the secure version of x86 exception frame as specified in Figure 3.2(b). This secure exception frame, instead of the original x86 exception frame, is generated on the operating system's kernel mode stack when an application running in an SP^3 domain gets interrupted. As shown in the figure, the first top 128 bytes of the secure exception frame represent the secure domain context , which is encrypted using a key private to the SP^3 system. This encrypted part contains the entire register context of the interrupted program. The SID value of the interrupted SP^3 domain is also saved at SID-0 to SID-3 field. SID value is stretched and then hashed to avoid overriding SID. The secure domain context is followed by the plaintext part, which is identical to the original x86 exception frame except for the zeroed EIP and ESP fields.

To generate this secure exception frame, we modified Xen's interrupt bouncer code, which handles forwarding of an interrupt to a guest operating system. Xen monitors every interrupt by intercepting it. If Xen decides to forward an interrupt to a guest operating system, it "artificially" creates an exception frame by writing to the kernel mode stack of the guest operating system, emulating the behavior of the CPU. This forwarding is implemented by the interrupt bouncer code, which we modified in such a way that if `current_sid` is not 0, it generates a secure exception frame instead of the standard one. At the moment Xen transfers control to the guest operating system, General-purpose registers (GPRs) are cleared and `current_sid` is set to 0.

To perform a return-from-interrupt on this secure exception frame, we defined a new instruction, called S_IRET. Executing this instruction causes traps to Xen via the invalid-opcode fault. We modified Xen's invalid-opcode handler to unwind the secure exception frame and resume the interrupted program. To restore the SP^3 domain context, Xen reloads GPRs and sets current_sid back from the saved values of the secure exception frame. The SP^3 paging extension takes advantage of this to correctly prepare a data structure when the operating system requests a page table update with a non-zero KID value.

A scheme is provided for the operating system and user applications to pass arguments and return values via GPRs. In this scheme, GPRs are normally cleared unless the cause of exception is a software interrupt; a user process can pass system call parameters via GPRs. The Type field tells whether GPRs have been cleared or not, indicating that the secure exception frame was generated by a software interrupt or another type of exception. Upon receipt of an interrupt–return request, Xen reloads GPRs from the saved register values unless the Type indicates the the secure exception frame was generated by a software interrupt, enabling a convenient channel for passing system call return values. Note that this facility does not necessarily incur leakage of information through GPRs, because applications can always clear contents of registers unused in the system call before generating a software interrupt.

3.4.2 Implementing SP^3 secure paging

During initialization, Xen reserves a pool of physical page frames for storing decrypted images. A page frame containing a decrypted image is mapped by PTEs with PFN value of the original page frame and non-zero KID field. It is important to recognize this class of PTEs with non-zero KID and the page frames mapped by them. Hence, we assign names for them to facilitate description. In the following discussion, we will refer to a page mapped with non-zero KID as an *SP^3 page* and the PTE for an SP^3 page as *SP^3 PTE*.

We use the P (present) bit of SP^3 PTE so that the processor can generate a non-present page-fault. These extra page-faults are intended to provide traps into Xen when accessing a SP^3 page needs attention of Xen, such as performing a check for PTE redirection. The page-fault handler of Xen is modified to separate this type of page-fault from other normal page-faults by examining the KID field of the PTE that caused the non-present page-fault.

Under the para-virtualizing architecture of Xen, this nontraditional usage of P can cause problems, since page tables are directly exposed to the operating system. We clear the P bit purposely even though the page is physically mapped by the operating system kernel. However, the operating system may be confused because it is possible for the operating system to see the P bit cleared when the bit was set before.

Without the hypervisor's shadow page table support, we could have only resolved this problem by modifying the operating system. However, Linux —our target operating system— already has a mechanism that can treat PTEs with P bit cleared as physically present. This facility fortunately enables us to avoid excessive modifications. In the current version of Linux, a page is considered non-present only if both P bit and PAT bit (bit 7) are cleared.[4] We exploit this by setting PAT bit for SP^3 PTEs so that Linux can recognize the page as present. Also, Linux does not experience any additional page-faults from this because Xen filters page-faults generated by SP^3 PTE.

When a page-fault is generated by SP^3 PTE, Xen fixes the fault by setting P bit with an appropriate value on PTE. Which page should be used is determined according to the SP^3 rule: if the current SID has access to the KID, Xen uses the decrypted image page. In other cases, the original page is used. In this process, the D (dirty) bit of the PTE is checked to synchronize between the two copies. The synchronization entails 4KiB AES operation which is time-consuming. However, under our lazy synchronization scheme, it happens only when it is needed. In practice, the synchronization is under full control of a user program (e.g., the program explicitly shares an SP^3 page with another SP^3 domain), or it occurs if the operating system wants to swap out the page to disk, which is rare in

[4] This facility is devised for memory regions mapped with PROT_NONE type. Linux clears P bit but sets PAT bit when loading a PTE for a page of that type. This way, the page is considered present by the kernel but the CPU generates a non-present fault upon access. Therefore, the kernel can raise protection violation, realizing PROT_NONE semantic.

modern platforms and already a very slow operation.

SP^3 PTEs have to be invalidated by clearing `P` bit whenever the domain is changed. This ensures the access permission of SP^3 pages to be reevaluated when the other SP^3 domain accesses that SP^3 pages. Once the SP^3 page is made present, access on the page will not generate any page-fault and the program can proceed. However, if the SP^3 PTEs' `P` bits are not cleared when `SID` changes, the other domain will access the old page, which can possibly contain a decrypted image. Therefore, this SP^3 PTE invalidation ensures the access permission of SP^3 pages to be reevaluated.

To implement this invalidation logic, Xen maintains a list of SP^3 PTEs that should be made non-present upon change of `SID`. When Xen reevaluates an SP^3 PTE by setting `P` bit, it also adds the PTE to the list. Later when `SID` changes, Xen goes through this list to clear the `P` bit, and the list is emptied. Exceptions and `S_IRET` can only change `SID`; the SP^3 PTE invalidation is performed when Xen handles those operations.

3.5 Evaluation

In our evaluation, we want to answer the following questions:

- How much performance degradation do SP^3 applications experience?
- How effective are the page-frame replication and the lazy synchronization?
- How does the performance overhead vary with applications' memory access patterns?
- What is the impact of SP^3 secure interrupt on performance?

To evaluate the impact of using SP^3 protection on performance, we first measured overall performance overhead with CPU- and memory-intensive workloads. Such a workload is chosen since our modifications are made on the CPU and the memory management part.

We then performed a micro-benchmark measuring the performance impact of the locality of the applications' page reference patterns.

3.5.1 Methodology

The machine used in our evaluation has a 3.2 GHZ Pentium 4 (HT) processor with 1 GiB of RAM. We used Xen version 2.0.4 and Linux kernel version 2.6.10, which is para-virtualized for Xen. Only a single virtual machine instance, namely dom0, is used for all experiments. Xen allocates 512 MiB of RAM for this guest virtual machine. For the Linux kernel setting, we used the default configuration in the original Xen distribution, which results in a uniprocessor kernel image without highmem support. We chose AES and RSA for our cryptographic primitives, whose implementation was taken from OpenSSL version 0.9.7e as C code without additional optimization.

We measured the performance of the SP^3 system by executing benchmark programs on a system running our SP^3 enabled Xen. This modified Xen is allocated an additional 256 MiB of RAM dedicated for storing decrypted images. For each benchmark program, two executables are generated from the same source code: one is an encrypted executable that can be executed only on the SP^3 enabled system, and the other is a normal insecure executable that can be used for performance comparison with a system without SP^3 protection. Both executables are statically linked with a modified version of dietlibc C library [35].

3.5.2 The price of protection measured in performance penalty

We wanted to know how much an application needs to pay for the SP^3 protection in terms of performance penalty. Since our SP^3 implementation changed the paging interface, we chose CPU- and memory-intensive workloads for measuring the impact on performance. For the workloads, we used programs from SPEC CPU2000 integer benchmarks.[5]

[5] Benchmark programs `252.eon` and `253.perlbmk` were excluded due to technical reasons. `252.eon` is written in C++ which is not supported in SP^3. `253.perlbmk`, a benchmark program based on the perl language interpreter, was very hard to port into SP^3 because of the huge system interface of perl language.

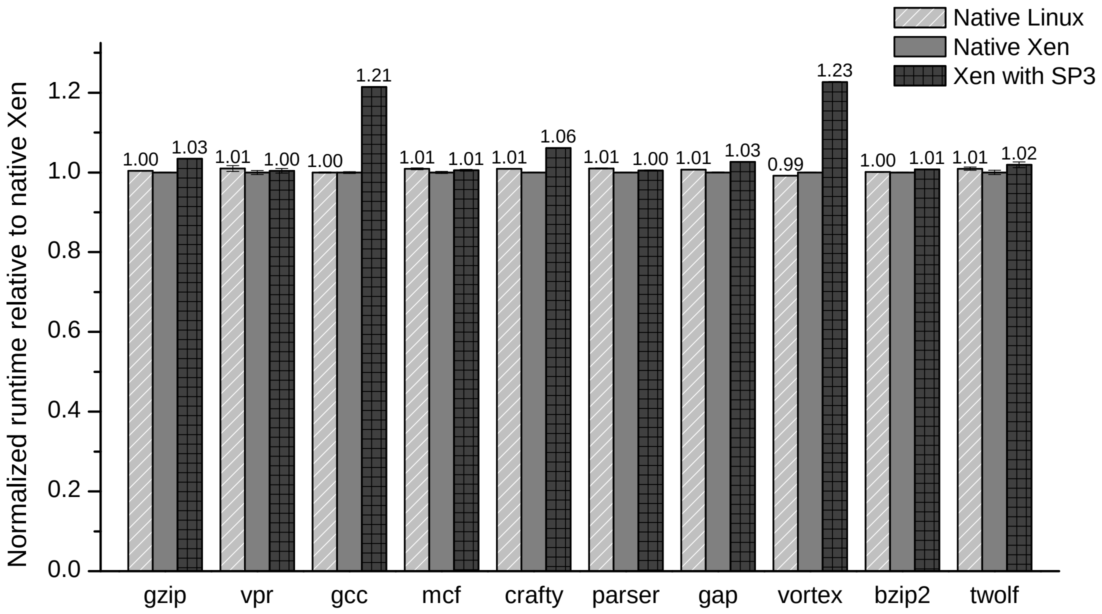


Figure 3.3: Results of SPEC CPU2000 benchmark measured on three different setups. The bars and the numbers on top represent the runtime of benchmark programs normalized to native Xen. Each value is the mean of 10 trials. Error bar represents one standard deviation.

Measuring the time to complete each program, we compared the running time of the workloads in three different setups. In the first setup, labeled as ‘Native Linux’, normal insecure executables were executed on native Linux without Xen. In the second setup, labeled as ‘Native Xen’, normal insecure executables were executed on Xen-Linux with the native, unmodified Xen. In the last setup, labeled as ‘Xen with SP^3’, the encrypted SP^3 executables were executed on the modified Xen-Linux with the modified Xen.

Figure 3.3 shows the benchmark results. The performance overhead is presented as a runtime increase normalized to native Xen. Overall, it takes 0-23% longer to finish the same program with SP^3 protection. While most benchmark programs suffer only 0-6% slowdown, only `gcc` and `vortex` programs pay unusually higher performance penalty. We found out that frequent system calls were the culprit of the anomaly of the two programs, which is detailed in Section 3.5.4.

The graph is normalized to the runtime of ‘native Xen’, not ‘native Linux’ because ‘native Xen’ shows the fastest result. This is counter-intuitive in that we expect native Linux to perform better due to the absence of the Xen layer. However, our experiment shows a slight ($\sim$1%) improvement when programs were running with Xen. Although we are not able to pinpoint the cause of this anomaly, this result is consistent with the results from other works in a similar setting [90]. The use of para-virtualized Linux or the inconsistent timer tick frequencies [105] could be contributing factors.

The performance result empirically confirms that both page-frame replication and lazy synchronization are indeed effective in reducing costly cryptographic operations. Since modified Xen keeps the copies of decrypted images, decryption is performed only when the image is initially created from the page that contains the original verbatim image. Once the decrypted image is created, it continues to be used without incurring any further decryption until there is a need to synchronize among images. However, this synchronization does not occur even after the application updates the decrypted image, thanks to the lazy synchronization. The update in a decrypted image propagates to the original page only when the operating system accesses the original page, which rarely happens because an operating system does not usually access application memory under normal conditions.

Since the overhead of page-wide encryptions is negligible, we can assume that the runtime penalty comes from the overhead of the PTE invalidation and subsequent page-fault for reevaluation. This type of penalty is paid less by a program with a small runtime footprint (i.e., accessing fewer pages during its activation between interrupts) than by one with a large footprint. If we assume that a statically larger program also has a larger runtime footprint, we can therefore expect that a statically larger program pays a higher penalty due to PTE invalidation than a smaller one. In fact, it is found that there is a positive relationship between the runtime footprint and the performance penalty. In Section 3.5.3, we present a more clear relationship between them with a micro-benchmark varying the size of dynamic memory footprint.

Securing interrupts can be another source of potential performance degradation. Although securing an interrupt involves cryptographic operations, in general it does not add much overhead because the frequency of interrupt is very low relative to the processor clock speed in modern computing systems, and also the overhead is overshadowed by the greater overhead of interrupt service routines and the resulting I/O operations. However, it is possible that secure interrupts can degrade performance of certain applications that request many simple system calls that the kernel can quickly return.

Secure interrupts are the culprit of the anomaly of the `vortex` and `gcc` benchmark programs. `vortex` is an object-oriented database whereas `gcc` is the GNU C compiler proper (i.e., `cc1`). By tracing system calls, we found that both programs generate a significant amount of system calls for mapping anonymous memory. These anonymous `mmap` requests are caused indirectly by dietlibc C library when it handles free allocation (i.e., `malloc` and `free`). Both benchmark programs belong to the software categories that heavily use free allocation: compilers and object-oriented databases. The overhead of system calls is analyzed in Section 3.5.4.

We were curious how the disk buffer cache affects performance since loading an executable from the disk carries initial decryption overhead in addition to the disk access overhead. We performed a comparison between 'cold' and 'hot' runs of the workloads. In the 'cold' run, we execute a workload program in boot-clean state without prior execution of the program, whereas in the 'hot' run, we execute the program right after executing the same program. When we measured and compared the two, we failed to find any significant difference. Although it is a non-trivial overhead to decrypt a page for creation of a decrypted image copy, it is obvious that the encryption penalty is hidden under the heavier overhead of disk I/O operation.

We also measured the performance of normal insecure executables running in the modified Xen. This was to find out the penalty on normal non-SP^3 applications when they are executed on an SP^3 enabled system. However, we could not detect any performance differences from the 'native Xen', and hence we do not report the result in the graph.

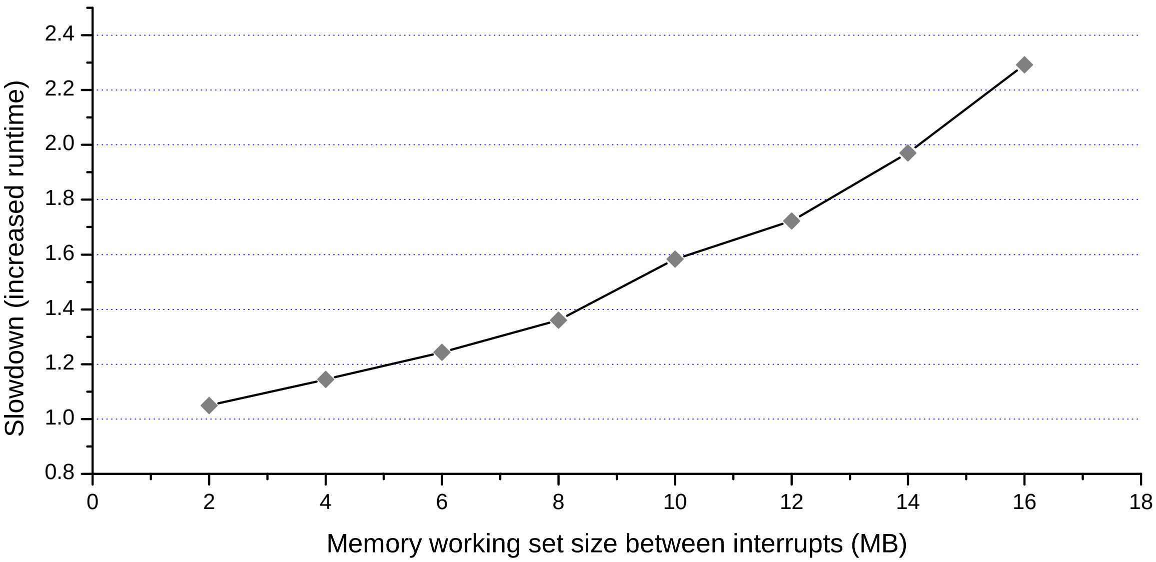


Figure 3.4: Impact of memory access locality on performance. Performance of a test program is shown as the size of working set of the test program increases. The y-axis shows increased runtime in SP^3 system normalized to the native Xen.

3.5.3 Impact of memory access locality

To obtain a clearer relationship between the runtime memory footprint and the PTE invalidation penalty, we performed a micro-benchmark with a varying runtime memory footprint size. The runtime memory footprint is defined as the number of pages – thus memory size – accessed by the program between each interrupt. The benchmark program varies its memory working set size and touches all of the memory pages continuously through a loop. In this way, we can artificially control the dynamic memory footprint.

Figure 3.4 shows the result. As expected, runtime penalty increases as the dynamic footprint – working set size between each interrupt – increases. If the dynamic footprint is small enough (less than 4MiB), the performance degradation is less than 15%. The performance penalty increase as the footprint increases, and when the footprint hits 14MiB, it takes twice as long.

Since many applications exhibit strong locality in accessing main memory, as can be seen in our SPEC benchmark, users of SP^3 system should not generally be concerned with

Platform	null	open
Native Linux	0.32 μs	2.07 μs
Native Xen	0.95 μs	3.27 μs
Xen with SP^3	10.6 μs	22.9 μs

Table 3.1: System call latency measured in microsecond.

performance degradation. Also this result is obtained from the un-optimized implementation: we did not aggressively optimize the invalidation and reevaluation logic. It is probable that we can further reduce the impact of invalidation on the performance by optimizing the invalidation logic. For example, we are considering invalidating a page directory entry instead of page table entry to reduce the number of entries in the invalidation list.

3.5.4 Impact of frequent system call

SP^3 applications that frequently request system calls are expected to suffer from the encryption overhead of SP^3 secure interrupt. To assess the increased cost of system calls in SP^3, we first performed a micro-benchmark that measures the overhead of system calls. We used the system call latency benchmark from `lmbench`, which was slightly modified to fit the SP^3 environment.

Table 3.1 shows the benchmark results. '`null`' measures the round trip overhead between user and kernel mode with minimum work required inside the kernel. '`open`' measures how long it takes to open and then close a file, thus more time is spent in the kernel.

As expected, the system call overhead is significantly higher in SP^3 compared to both native Xen and Linux. This increased latency is due to the increased round trip time for user/kernel crossing, which is caused by the encryption of SP^3 secure interrupt frame. This result also confirms the slowdown of `gcc` and `vortex` benchmark programs in Section 3.5.2, both of which make system calls at the average rate of 2,300 requests per second.

To better understand the relationship between frequent system calls and performance, we also measured the rate of system calls made by each of the SPEC CPU2000 benchmark

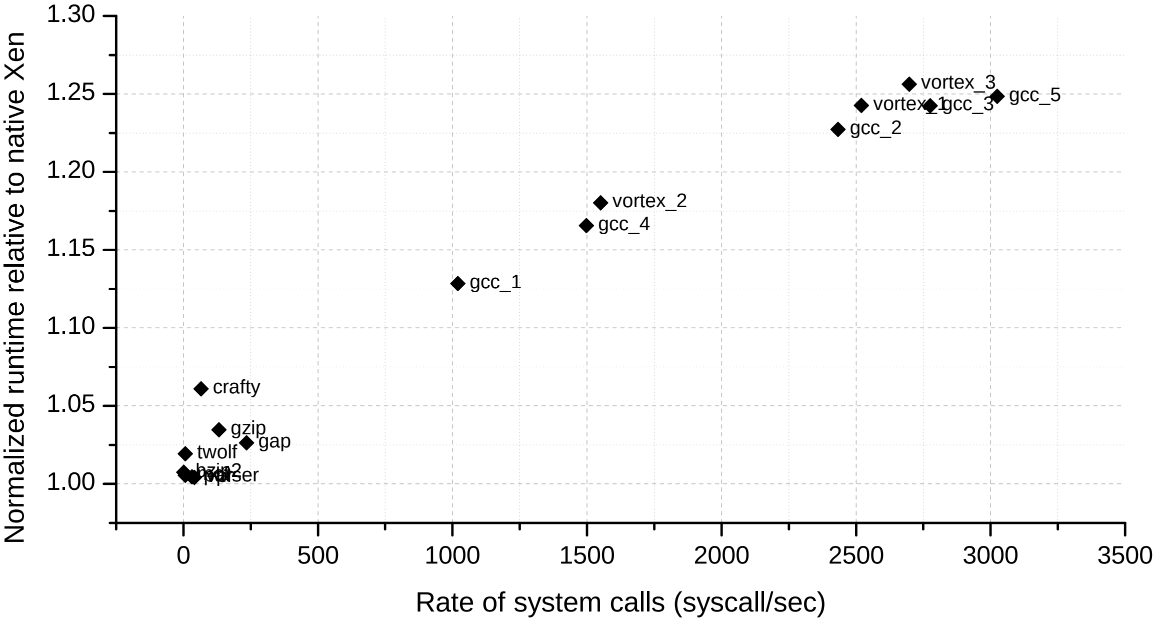


Figure 3.5: Impact of frequent system calls in SPEC benchmark on performance. Sub-benchmarks of `gcc` and `vortex` are individually labeled. Each value is the mean of 10 trials.

programs. We counted the total number of system calls of each program and the total counts were divided with the runtime in the native Xen setting. The total counts rarely varied because the benchmark programs are strictly deterministic.

Figure 3.5 shows the relationship between the average rate of system calls and the increased runtime of each benchmark program. To provide a finer-grained result, we broke down `gcc` and `vortex` benchmarks into their sub-benchmarks, which are individually labeled with a number suffix.

As can be seen, there is a positive relationship between the rate of system calls and the runtime penalty. The figure shows that every benchmark program with more than 10% penalty makes more than 1,000 system calls per second. On the other hand, the programs with low system call rate are clustered at the very low end of performance overhead range. Given that 1,000 syscall/sec is considered unusual for real production applications[6], we can

[6] The high rate of system calls is caused by handling of free allocation of dietlibc, which is optimized not on performance but on its size.

expect less than 10% performance penalty for typical applications.

3.6 Related work

Virtual machine research has a long history. In his thesis [46], Goldberg in 1972 laid the theoretical groundwork on virtual machine monitors, including the definition of virtual machine and the conditions that hardware must meet in order to be virtualized. Following this work, requirements for virtualizable architecture were formally analyzed by Popek and Goldberg [80]. Virtualization was a commercially successful technology in the '70s [47], with IBM VM/370 [88] being the prominent system.

It is safe to say that recent resurgence of virtualization is largely initiated by the commercial success of VMware, whose origin can be traced back to Stanford SimOS [85] and Disco project [24, 25]. VMware ESX server [107] runs directly on top of the hardware without the need for a host operating system. This architecture is also called bare-metal architecture. On the other hand, VMware Workstation [97] requires a host operating system. Waldspurger [107] details optimization algorithms and strategies for memory management implemented in the VMware's hypervisor. Sugarman [97] presented I/O architecture of VMware Workstation. Later, the open-source Xen hypervisor from Cambridge [13] stimulated both academia and industry for virtual machine research. Fraser [41, 42] detailed Xen's second generation I/O architecture.

Robin [84] analyzed the Pentium's ability to safely support a virtual machine monitor and found that the general x86 system architecture does not meet the virtualizability conditions. With respect to the lack of virtualizability, the initial commercial success of the VMware can be attributed to the dynamic binary translation, which can detect and trap offending instructions. Bruening has refined the dynamic optimization work addressing adaptation [21] and code cache management [22]. QEMU [15] is an open-source machine emulator based on dynamic translation techniques. Later, Intel and AMD extended their hardware to support virtualization, largely eliminating the need for dynamic translation or para-virtualization. Adams published a comparison work on software-based versus hardware-based support for x86 virtualization [8].

With this background, virtual machine monitors are being utilized to solve many system security problems. IntroVirt [59] used virtualization to log/replay system events, achieving a perturbation-free intrusion detection system that can also detect past intrusions. Garfinkel and Rosenblum [44] designed an intrusion detection system based on virtual machine introspection. The proposal of using hypervisor in commodity mobile systems [32] is motivated by the advantage of using hypervisor for implementing security services. King's backtracking intrusion [62] uses logging and replaying of operating system events to track down the vulnerability and entry point of attacks. Denali isolation kernel [108] uses a heavily modified guest operating system to facilitate performance and scalability of virtual machine monitors.

Ta-Min's Proxos [101] is a hypervisor-based trust-partitioning system in which users can configure the trust on the operating system. A trusted application runs in a private trusted operating system created by the underlying hypervisor. A set of system calls, which the user can specify, is dynamically forwarded into another operating system instance, which is a full-fledged operating system but untrusted. In contrast, our system provides protection to user memory on a per-page basis, and does not require any private operating system instance.

3.7 Summary

This chapter detailed the major implementation of this thesis: implementing the SP^3 model in the x86 architecture by modifying the Xen hypervisor. The extension to the x86 system architecture was first specified to apply the SP^3 protection model defined in the previous chapter. Then the Xen hypervisor is modified to emulate the extended x86 architecture. The modified Xen serves as the new trust base for the user applications. The user applications do not have to trust or rely on the operating system for the secrecy of the user information.

We use *page-frame replication* to reduce the number of cryptographic operations by keeping decrypted versions of a page frame. We also employ *lazy synchronization* to minimize overhead due to an update to one of the replicated page frames. Our evaluation result of the modified Xen shows that it increases the application execution time by 0-23% for CPU and memory-intensive workloads.

CHAPTER IV

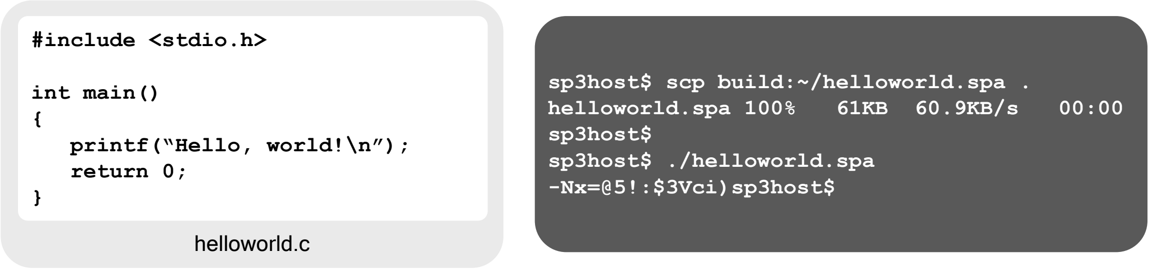
PROGRAMMING ENVIRONMENT IN SP3

4.1 Introduction

In this chapter, we explore the impact of introducing SP^3 to the programming environment and the problems in application construction and operating system constructs. We present solutions to the specific problems in operating system components and application programming environments such as Application Binary Interface (ABI), C runtime library, program compilation and linkage, program loader, memory manager, and signal delivery mechanism.

These problems originate from the fact that SP^3 changes the underlying memory management semantics in a number of ways. These changes result in a new programming environment, in which applications and the operating system must beware of the constraints imposed by SP^3. In this environment, an application programmer must specify rules by which data should be shared with the operating system. The operating system is no longer capable of obtaining meaningful information from protected memory pages.

This chapter starts with an example program in Section 4.2. Section 4.3 presents the process of generating an encrypted executable and loading the executable into the memory. Section 4.4 details how an operating system can be made to support SP^3 memory and interrupt semantics. Section 4.5 considers the application programming in the SP^3 environment by discussing how application developers can easily take advantage of SP^3 protection.



```
#include <stdio.h>

int main()
{
   printf(“Hello, world!\n”);
   return 0;
}
```

```
sp3host$ scp build:~/helloworld.spa .
helloworld.spa 100%   61KB  60.9KB/s   00:00
sp3host$
sp3host$ ./helloworld.spa
-Nx=@5!:$3Vci)sp3host$
```

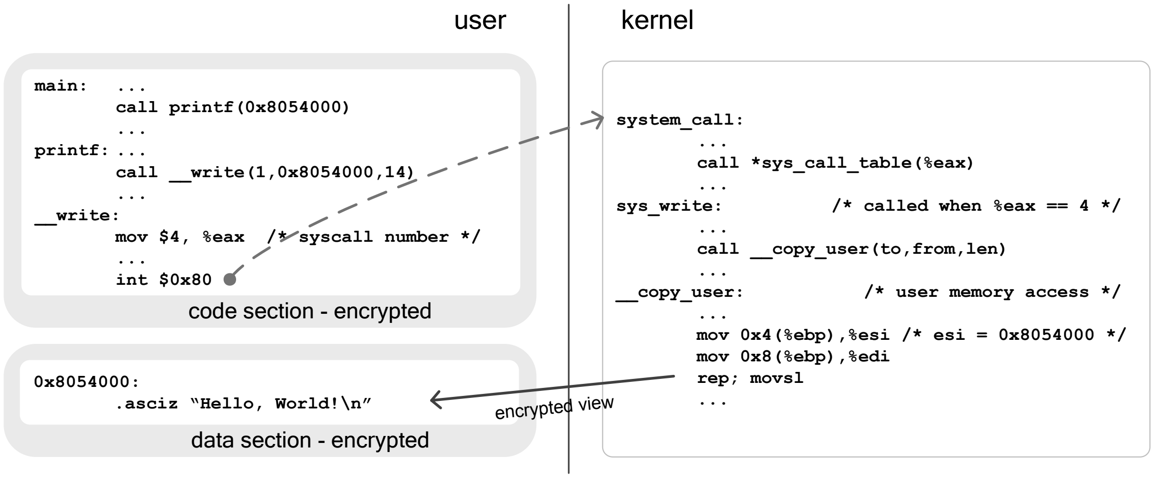
Figure 4.1: The first, but incorrect, hello-world SP^3 program. Shown on the left is the C source code, and on the right is the result of the program. The program prints out a garbled string instead of the hello-world string. This is caused by the fact that the operating system is accessing the string constant in the user space, which is rendered as encrypted when it is viewed by the operating system.

4.2 Hello world program in SP^3

Programming issues in SP^3 can be best illustrated by the familiar “Hello world” C program and a discussion of why this simple program fails to deliver the expected result. In this section, we examine what happened, what went wrong, and how to solve this problem to get the correct result. Throughout the examination, we gain some insights into what it would be like to write a program in the SP^3 environment. It should be remembered that a programmer can no longer assume the operating system is capable of reading meaningful information from user space: the programmer has to explicitly put the information to be shared in the unencrypted region of memory.

Figure 4.1 shows on the left a typical C code that prints out *Hello, world!* to the standard output, and on the right is the result of the program's execution in an SP^3 capable host running Linux. The code is compiled, linked, and post-processed to generate a valid SP^3 executable, which contains encrypted code and data segments[1]. Encrypted versions of the executables are given the filename extension of `.spa` to distinguish them from unencrypted, hence insecure, versions of executables. Shown on the right of the figure is the screen output of `sp3host` on which the hello-world SP^3 executable is executed.

[1] The details of generating encrypted executable are discussed in Section 4.5



```
main:   ...
        call printf(0x8054000)
        ...
printf: ...
        call __write(1,0x8054000,14)
        ...
__write:
        mov $4, %eax  /* syscall number */
        ...
        int $0x80
```

```
0x8054000:
        .asciz "Hello, World!\n"
```

```
system_call:
        ...
        call *sys_call_table(%eax)
        ...
sys_write:          /* called when %eax == 4 */
        ...
        call __copy_user(to,from,len)
        ...
__copy_user:            /* user memory access */
        ...
        mov 0x4(%ebp),%esi /* esi = 0x8054000 */
        mov 0x8(%ebp),%edi
        rep; movsl
        ...
```

Figure 4.2: Control flow of the first hello-world program when `printf` is executed. `printf` calls `write` system call, which makes transition to the kernel space. The kernel service routine of `write` system call eventually copies from the user space data for its terminal I/O.

The result is surprising because this simple example does not seem to produce the correct result: it prints out random characters to the standard output. In fact, it turns out that the random characters are actually an encrypted text of the *Hello, world!* string.

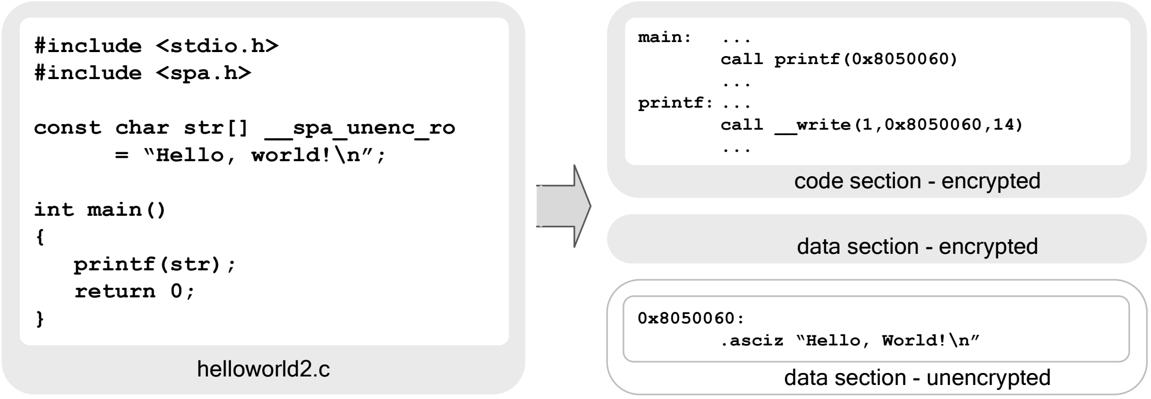
We can easily realize that the operating system is reading from the user memory that contains the string, and the memory appears as encrypted from the viewpoint of the operating system. This is actually the exact behavior that is expected in the SP^3 system: the operating system should see the verbatim – hence encrypted – image whenever it accesses protected user memory. Since the operating system is indifferent to the content of what it reads, it uses the encrypted text for the actual terminal I/O operation.

It is more revealing to investigate what is going on under the hood. In Figure 4.2, the code and data of the program is shown in the form of a binary image loaded into memory. Labeled are the entry points of relevant functions and the location of the *Hello, world!* string. The figure also illustrates the switch to the kernel space as the result of calling `printf` function. The `printf` C library function consists of two parts: creating a human-readable formatted character string and writing the result to the standard output. The second part is essentially equal to making a UNIX write system call. The write system call takes three arguments: a file descriptor that references the file to be written to, a pointer to a buffer that contains data, and the number of bytes to write. The user code section in Figure 4.2 shows that the printf eventually invokes write on the the standard output (file descriptor 1) with the pointer to the string (0x8054000) and the length of the string (14). The actual transition to the Linux kernel is triggered by the software interrupt (int $0x80) with syscall number 4 in the eax register. Note that in SP^3, the current SP^3 domain changes to 0 by this software interrupt. The right of the figure shows the kernel service routine for handling write system call. At the system call entry point, the syscall dispatch table is looked up and sys_write is called. To obtain the data to write, sys_write has to read from the user memory at the location 0x8054000 (via __copy_user), but it reads an encrypted version of the hello-world string since the current domain is 0. This encrypted string is then used for terminal I/O, hence the garbled output that appears in the terminal.

The actual problem with this program is that the specification – the code – does not match the programmer's intention: the programmer wants the plaintext to appear in the output, but the program is one that writes encrypted text to output. Note that this program is *working correctly* in the sense that it does what it is specified to do, and also is *legitimate* in the sense that its specification does not violate any explicit or implicit rules. If the programmer had wanted to write the encrypted text (e.g., `$ ./helloworld.spa > secret_message.txt`), then the current hello-world program would be a correct program.

Figure 4.3 shows an improved version of the hello-world program that reflects the programmer's intention of writing plaintext to the screen. Noticeable in this program are the non-standard C constructs: the inclusion of header spa.h, and the __spa_unenc_ro modifier. __spa_unenc_ro is defined in spa.h to actually expand to a GCC C extension[2] that allows specifying the name of the object file section in which the annotated variable will be allocated. Although this is a non-standard C extension, Microsoft Visual C compiler

[2] Defined as `__attribute__((section(".rodata.unenc")))`



```
#include <stdio.h>
#include <spa.h>

const char str[] __spa_unenc_ro
      = “Hello, world!\n”;

int main()
{
   printf(str);
   return 0;
}
```

```
main:   ...
        call printf(0x8050060)
        ...
printf: ...
        call __write(1,0x8050060,14)
        ...
```

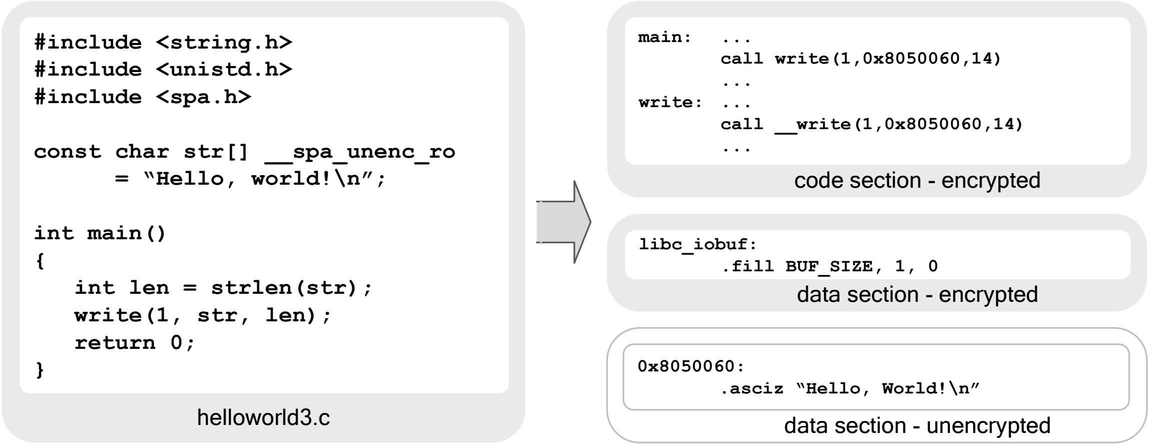
```
0x8050060:
        .asciz “Hello, World!\n”
```

Figure 4.3: The second version of our hello-world program. The __spa_unenc_ro modifier places the string in the unencrypted region of data section.

– another major production compiler – also supports a form of compiler extension that performs an equivalent function.

By specifying with the __spa_unenc_ro modifier, the C string constant is now defined to be placed in a special data section that is not to be encrypted. When the program is being linked from object files, the linker makes sure the data from the special section are not encrypted. When the program is being executed, the binary loader of the operating system maps the unencrypted section in the virtual address space with null KID, so that the pages are viewed as unencrypted images both by the operating system and the application. The result of the program loading is shown on the right of the figure, illustrating the memory layout of the program with the hello-world string being placed in the unencrypted data section.

Although this program seems to be working, this program only works correctly because of the way the printf C library function is implemented, which is one subtle yet important problem. Since printf creates a formatted character string, some C library implementations might use an internal buffer to temporarily store the formatted string, and then use this buffer for the subsequent write system call. If the C library of this program uses this internal buffer, and the buffer is located in the encrypted data section, the program would



```
#include <string.h>
#include <unistd.h>
#include <spa.h>

const char str[] __spa_unenc_ro
      = "Hello, world!\n";

int main()
{
   int len = strlen(str);
   write(1, str, len);
   return 0;
}
```

```
main:   ...
        call write(1,0x8050060,14)
        ...
write:  ...
        call __write(1,0x8050060,14)
        ...
```

```
libc_iobuf:
        .fill BUF_SIZE, 1, 0
```

```
0x8050060:
        .asciz "Hello, World!\n"
```

Figure 4.4: The final, and safe, version of hello-world program. The call to the `printf` is replaced with direct call to `write` system call. This is to eliminate the uncertainty coming from the implementation specifics of the C library.

fail again. Fortunately, the C library of this program happens not to use the internal buffer for the I/O operation.

Still, however, we cannot be sure whether the behavior of this C library is predictable and consistent: our program might just hit a lucky case in which `printf` is using the user string directly because the string requires no formatting. The `printf` might be implemented to optimize on the no-formatting case, which can be detected by examining the number of arguments. If this is what is happening, relying on this behavior would be an unsafe practice.

Therefore, unless a C library with well-defined behavior is being used, we should avoid using `printf`. Figure 4.4 shows the final and safe version of our hello-world program, which replaces `printf` with a direct invocation of `write` system call. This version is safe because we know for sure that the string will be directly accessed by the operating system. Shown on the right is the memory layout of the program, indicating both the C library buffer and the hello-world string to highlight the difference in the data location.

From these examples, it becomes clear that finding out the correct *location* of each data element is the unique activity that determines the correct outcome of SP^3 programs. If a programmer intends for a data element to be seen as plaintext from outside, he should place it in *public*, meaning the data element should be located in the unencrypted section. If he wants the data element to appear as encrypted from the outside, he should place it in *private*, meaning the data element should be located in the encrypted section.

It is fundamentally the programmer's job to determine which data to put in public and which data to keep in private. As can be seen in the hello-world examples, the correctness of the program ultimately depends on the programmer's intention, and the true intention is impossible to infer from the program. Therefore, SP^3 only provides tools and interfaces for writing SP^3 programs easily, and does not attempt to solve the problem of how to automatically "transform" an insecure application into a secure SP^3 application. In this respect, the approach of SP^3 differs from that of Overshadow [28], since Overshadow implicity assumes that someone other than the application writer can somehow figure out the intention and then secure unmodified, insecure applications.

In summary, the failure of the original hello-world program is caused by the positional semantic dependency of the user space pointer passed to the kernel during a system call. Therefore, applications have to be careful when making system calls or invoking C library functions that can potentially let the operating system refer to user space memory. In addition, as much as the application and C library have to be concerned about having the operating system access user space, the operating system itself must take great care in accessing user memory.

We conclude this section with a list of observations made from analysis of the hello-world programs. The conclusion given here applies to general SP^3 applications, and serves as the motivation of the work presented in this chapter.

- The programmer must decide which data to put in public and which data to keep in private.

- Programs must be able to specify the location of each data element.

- Linker and loader must be able to differentiate encrypted and unencrypted sections.
- Some C library functions must be either avoided or made SP^3-safe.
- When making system calls, applications must beware of the consequences of passing user pointers.
- The operating system must take great care in accessing user space memory.

4.3 SP^3 executable

In many cases, little attention is given to the subjects of the internal structure of program binaries and the procedure of constructing them. This is because application programmers usually do not have to know the underlying details about program linking and executable formats. Besides, these subjects already have mature solutions that are also well-standardized, so there is little merit in changing the established procedures.

However, SP^3 requires a non-trivial investigation into the build procedure in order to address SP^3 specific requirements. In SP^3, program data should be separated into either one group of data that should be encrypted or the other group that should not be encrypted. Accommodating this requirement entails a form of support from the program building procedure and the executable format.

In this section, we present our solution to this problem of creating encrypted SP^3 executables and loading them into the system. To attain maximum compatibility, our solution utilizes existing standards and techniques. An encrypted SP^3 executable can be generated using a standard suite of compiler and linker without modification. The executable can be loaded by the ELF binary loader of Linux, which is slightly modified to take care of the encrypted segments of the executable.

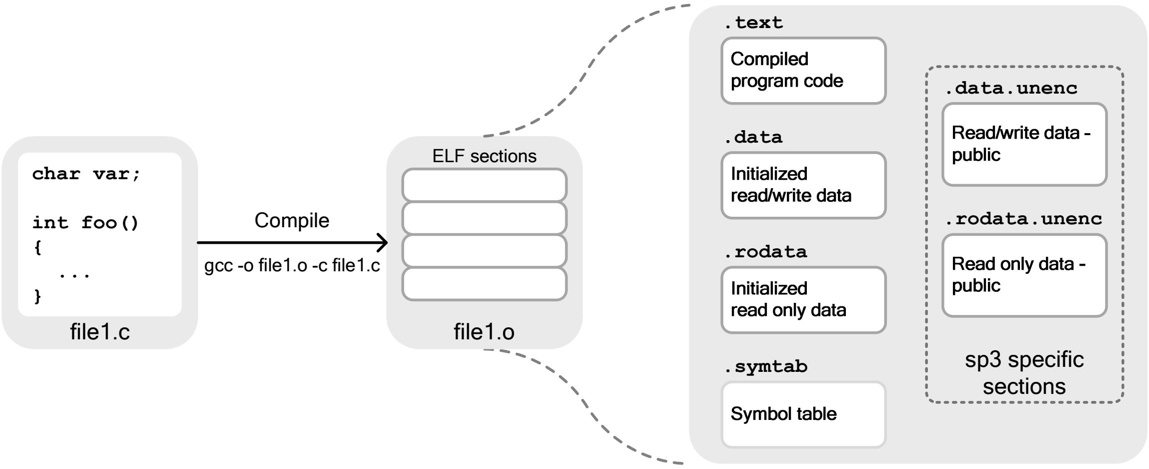


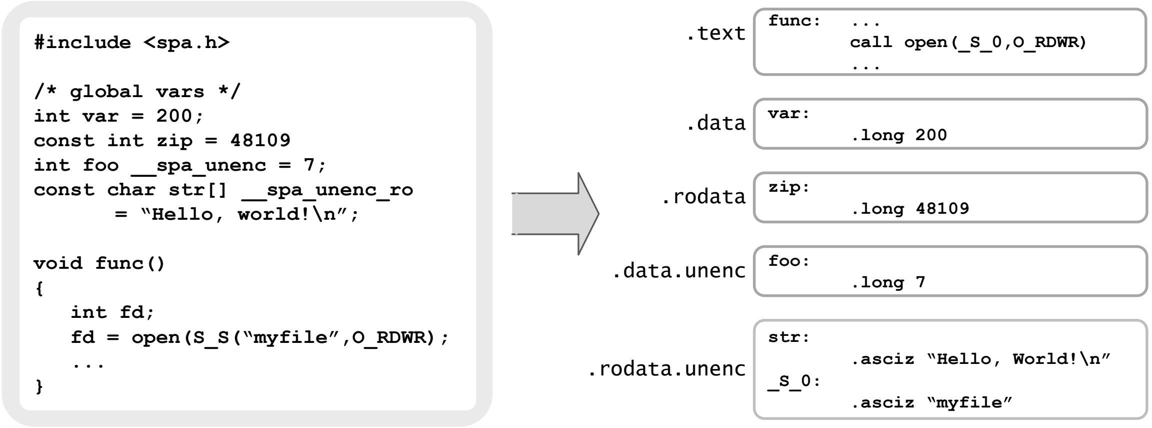
Figure 4.5: Various sections of an object file. A conventional ELF object file consists of several sections that contain program code, writable data, read-only data, and a symbol table. To support SP^3 programs, two additional sections are defined.

4.3.1 Creating an SP^3 executable

The direct outcome of compiling a source code file is an object file. An object file contains the compiled code – usually in the machine language of the target system – and the data defined in the source code. Later, in a phase called program linking, multiple object files are then collected and packaged into an executable file that is loadable by an operating system for execution.

An object file typically consists of multiple *sections*. In an object file section, data elements with the same attribute are grouped together and made into a block of binary data. Each section is identified by a section name, and later in the linking phase, a linker uses section names to collect and merge sections from multiple object files. The final, loadable unit of combined sections is known as a *segment*[3]. For example, the section name `.text` identifies the section that contains compiled program code. To create an executable, the linker finds all `.text` sections from object files and then merges them into a segment of binary code, which can be easily mapped into process's address space by a program loader.

[3] This is a linker terminology and should not be confused with x86 segmentation.



```
#include <spa.h>

/* global vars */
int var = 200;
const int zip = 48109
int foo __spa_unenc = 7;
const char str[] __spa_unenc_ro
      = “Hello, world!\n”;

void func()
{
   int fd;
   fd = open(S_S(“myfile”,O_RDWR);
   ...
}
```

```
.text           func:   ...
                        call open(_S_0,O_RDWR)
                        ...

.data           var:
                        .long 200

.rodata         zip:
                        .long 48109

.data.unenc     foo:
                        .long 7

.rodata.unenc   str:
                        .asciz “Hello, World!\n”
                _S_0:
                        .asciz “myfile”
```

Figure 4.6: An example SP^3 object file made from an annotated C source code. The compiled code and variables without annotation are placed in the conventional ELF sections. The annotated variables are placed in `.data.unenc` and `.rodata.unenc`, which are SP^3 specific.

Figure 4.5 illustrates various sections that can be found in a typical ELF object file, as well as sections that are specific to SP^3 applications. Typically, a compiler treats writable data and read-only data separately, and places them into the `.data` section or `.rodata` section. This separation is necessary in order to facilitate memory protection, memory sharing and the copy-on-write scheme found in most operating systems. The `.symtab` section contains a table of symbols that are exported/imported by the object file. The linker uses this symbol table to calculate relocated addresses when sections are merged.

The two SP^3 specific sections, `.data.unenc` and `.rodata.unenc`, are similar to `.data` and `.rodata`, but they are different in that these sections contain data elements designated as public in an SP^3 application. Therefore, the content of these sections will not be encrypted when the final SP^3 executable is generated.

To place a data element in one of the two special sections, we utilized a C compiler extension that allows specifying the section name in which the variable is defined. This way the programmer can easily annotate public data elements in the source code, as already demonstrated in our second hello-world program in the previous section.

An example of annotated source code and the resulting object file is illustrated in Figure

4.6. The first two global variables `var` and `zip` are defined without annotation; therefore they are placed in the `.data` and `.rodata` sections respectively. The next two variables are annotated with `__spa_unenc` and `__spa_unenc_ro`, indicating they are to be rendered unencrypted even when viewed from outside. The annotations instruct the compiler to place the variables `foo` and `str` in `.data.unenc` and `.rodata.unenc` sections, respectively. In the function, the string constant used as the argument for the `open` system call is enclosed with `S_S()`. Note that we want the operating system to read the file name as unencrypted string. This expression is a convenient way of placing a string constant in the `.rodata.unenc` section.

Except for the annotated static variables, all other data elements are to be encrypted. This is also true for the variables created in the runtime. In SP^3, the program stack is mapped with SP^3 protected pages; therefore automatic variables created in the program stack are encrypted when accessed by the operating system. The program heap – free-allocated memory – is also mapped with SP^3 protected pages. In a nutshell, everything in the process address space will be protected except for the annotated static variables and the memory allocated at runtime with anonymous `mmap` with null KID, which is the topic of Section 4.4.

Having described how to differentiate encrypted and unencrypted sections in an object file, we turn our attention to the whole procedure of generating an encrypted SP^3 executable. Depicted in Figure 4.7 is an example case of building an SP^3 executable using our build system implemented on Linux. Our build system utilizes the standard GNU linker and a couple of perl scripts. At the center, the linker produces an intermediate executable file (`prog`) from object files and a C library[4]. Using a special linker script (`spa_lds.ld`), the linker collects and merges sections from object files including the SP^3 specific sections. The linker script also makes sure that all mappable segments are page-aligned. The intermediate executable file then undergoes a post-processing phase with the help of a perl

[4] The C library is modified to adapt to SP^3 environment

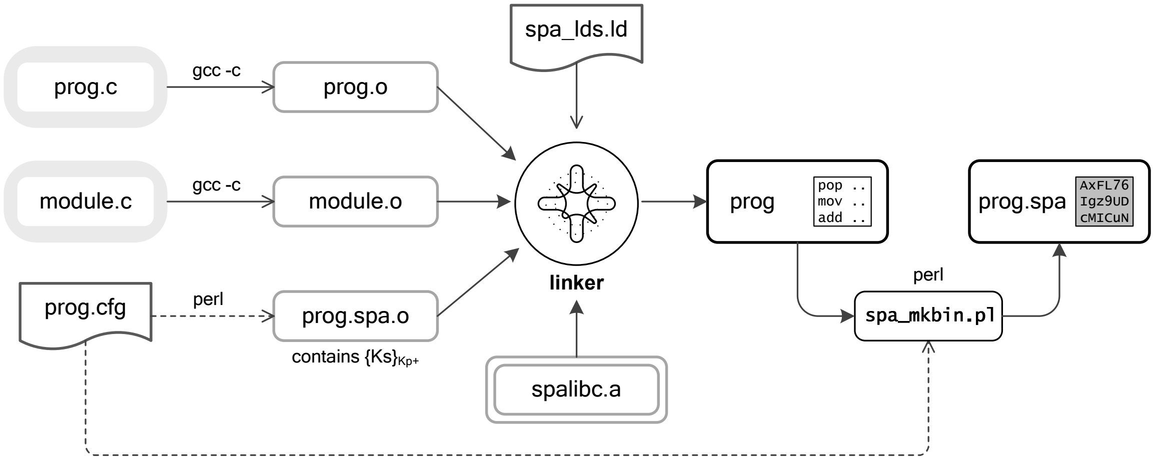


Figure 4.7: A procedure for building an SP^3 executable. With the help of a special linker script, the linker is able to collect and merge SP^3 specific sections from the source object files. The resulting executable is then post-processed to produce the final SP^3 executable, which contains encrypted segments.

script (`spa_mkbin.pl`). This post-processing phase is where the actual encryption with K_s takes place. The `.text`, `.rodata`, and `.data` segments are encrypted in the final SP^3 executable (`prog.spa`). The perl script finds the value of the key K_s from a configuration file (`prog.cfg`). This configuration file serves as the central place that specifies all the cryptographic variables such as K_s and K_{P+}. The configuration file is also used to generate a special object file (`prog.spa.o`) that contains SP^3 specific metadata information containing $\{K_s\}_{K_{P+}}$.

In the above build procedure, two valid executables are produced: `prog` in the example is the executable without encrypted segments, and `prog.spa` is the executable with encrypted segments. Certainly, a system without SP^3– vanilla Linux, for example, – is capable of running `prog` but not `prog.spa`. In a system with SP^3, both `prog` and `prog.spa` can be executed correctly, provided that the public key of the SP^3 matches with the one used in the `prog.spa`. Note that the two binaries are functionally equivalent with the only difference being the capability of running on an SP^3 system. This is useful in debugging and performance comparison, and these executable pairs were used in the performance evaluation of the Xen-based SP^3 system in Chapter III.

4.3.2 Loading an SP^3 executable

When an executable file is being executed, the operating system invokes a binary loader module that interprets the specific executable format. The binary loader then maps the executable segments into the process's address space, and sets up program arguments and environment variables. Finally, the loader prepares for the initial entry into the program by reconfiguring the in-kernel data structure associated with the process, making the process scheduled just like any other processes.

To run SP^3 executables, the binary loader must be changed to support for the SP^3 executables. The binary loader must be able to understand the format, identify encrypted segments, and construct an initial condition in a way that is not intrusive to the user space memory.

We implemented our SP^3 binary loader by modifying the ELF binary loader of Linux as well as the kernel handler for `exec` system call. The modified binary loader detects an SP^3 executable by examining a flag in the ELF file header. This informs the loader of the need to create an SP^3 domain and the presence of encrypted segments in the executable. A new SP^3 domain is created by executing `Alloc` instruction, which invokes the underlying SP^3 system that extracts K_s directly from $\{K_s\}_{K_{p+}}$, and sets up permissions accordingly. To map the segments into the address space, Linux utilizes the memory file map by invoking `mmap` on each of the segments of the executable file. We modified the loader in such a way that we call `mmap` with a special parameter specifying the KID value for the address range of the encrypted segments. Then the loader allocates memory for the program stack and then copies command line parameters and environment variables. The loader maps the stack with null KID only for the region of memory that contains kernel-supplied information. The rest of the the stack is mapped with a valid KID.

The changes in the program loading also affect the Application Binary Interface (ABI) regarding the initial program state presented to the application at the entry point. For instance, the System V ABI [5] strictly defines the initial contents of the registers including the stack pointer. However, in SP^3, the operating system cannot arbitrarily set the stack pointer, which must be set to the top of the stack, whose address might vary every time the program is executed. We solved this problem by delivering the stack address in a general purpose register. This change requires the program to load the stack pointer from a general purpose register at the point of program entry.

The C library that is supplied as a part of the SP^3 application development package provides an entry point function that handles all the changes in the ABI. The entry point function also implements the user level dispatcher for the signal, which is discussed in Section 4.4. The entry point function is listed in Appendix B.1.

In our implementation, a new SP^3 domain is created when `exec` is called. Therefore, in a system with explicit separation between creating a process (i.e., `fork`) and overlaying memory with executable (i.e., `exec`), SP^3 domains kept in the underlying SP^3 system do not correspond to the processes of the operating system. Rather, the domains are associated more with particular instances of SP^3 executable files.

Although it seems more easy and natural to design a protection system built around the concept of process, applying that approach would produce a system that is very complex and hardly practical. This is because there is a logical and temporal gap between `fork` and `exec`: if a protection domain is created upon `fork` when the actual secured content is loaded upon `exec`, a complex scheme has to be constructed to secure the domain in between against an untrusted operating system. For example, the Xom/Xomos architecture [66] suffers from this because Xom equates its concept of protection domain (Xom compartment) with the process of operating system. As a result, many of the artifacts in Xom architecture are dedicated to solving corner case problems in creating and maintaining Xom compartment. In addition, Xom needs to deal with how to securely set individual memory bytes or registers at runtime after the compartment/process is created.

SP^3 defines its protection domain to be a completely independent notion, which is a conscious design decision that we discussed in Chapter II. The independence from any operating system abstractions not only promotes a clean and simple interface, but also permits flexible use of the SP^3 domains. For example, a single SP^3 domain can be created for a group of processes executing the same SP^3 executable. With this approach, we can save underlying SP^3 resources. But in our current implementation, we create a new SP^3 domain each time `exec` is called even if there is another process instance of the same executable, which is also a valid approach.

4.4 Operating system

SP^3 introduces a number of changes to the underlying interface that operating systems are built upon. To adapt to these changes, a conventional operating system must be modified in order to correctly load user applications, to support the extension of the paging interface, and to make the operating system function correctly. In the previous section, we already showed how Linux's ELF binary loader has to be changed to support SP^3 executables. In this section, we discuss the changes made in other areas of the operating system.

Before we move on to the detailed discussion, it should be emphasize that these modifications are never necessary conditions for the security of SP^3 protection. In other words, these modifications are just to achieve correct functionality of the operating system; a malfunctioning operating system can crash the system, but nevertheless, it cannot break the privacy of applications.

4.4.1 Memory management with SP^3 paging

An operating system must take into account the new KID field of the page table entry (PTE) structure of SP^3 enabled hardware. Obviously, this affects memory management routines that handle pages and address-spaces. Kernel routines that directly interface with the hardware paging unit, such as the page-fault handler, have to recognize the KID field.

Kernel routines that manage the virtual address space of each process also have to differentiate regions of virtual addresses mapped with different KID values. In addition, the operating system must provide a KID-aware memory mapping facility, such as `mmap` system call, to user applications.

It is helpful to know a bit on Linux's memory management internals. In Linux, the internal harness for memory management consists of a low-level paging subsystem and a high-level address-space management subsystem. The paging subsystem deals primarily with page allocations and subsequent updates to PTEs on a per-page basis, whereas the address-space management subsystem keeps track of the usage of each process's virtual address space on the basis of a region of virtual address.

The paging subsystem of Linux is mostly implemented as part of page-fault handling routines. This is because of the use of demand paging and the copy-on-write scheme, which makes it necessary to update PTEs during the handling of page-faults. This means that a physical page for a virtual address is mapped later when a page-fault occurs on that virtual address, not earlier when the virtual address is prescribed for a particular use by the address-space management subsystem. When the page-fault handler tries to map the faulting page, the handler consults the address-space management subsystem to find out the memory attribute associated with the fault address, and specifies PTE's Writable (`W`), Executable (`X`), Dirty (`D`), and Present (`P`) bits accordingly.

The address-space management subsystem maintains the memory attribute information using the in-kernel structure called `vm_area_struct`. Each `vm_area_struct` descriptor represents a consecutive, non-overlapping region of virtual address of an address space. The descriptor includes a great deal of information: start and end addresses of the region, the memory attributes such as read/write/executable flags, the file descriptor if the address region is file-mapped, and so on. Linux organizes `vm_area_struct` descriptors in such a way that each address region shares the same memory attributes. Therefore, Linux implements a merge/split logic on address regions: multiple adjacent regions are merged when the regions share the same attributes, and a region is split when a part of the region has its attribute changed. The address-space management subsystem is consulted by the paging subsystem when a page-fault occurs, and the page-fault address is used to search for the matching vm_area_struct descriptor. To facilitate a fast search, Linux maintains a balanced tree of vm_area_struct descriptors for each address-space.

The mmap system call of Linux serves as the most important facility that allows a user process to allocate memory in its virtual address space. mmap has a very versatile interface: the mmapped memory can be a free memory with no strings attached (i.e., anonymous map), or associated with a file (i.e., file map or named map) so that the file can be accessed through virtual address space; the mmapped memory can be either private or sharable; the attributes of the mapped memory can also be specified. This versatility makes mmap capable of satisfying almost every address-space related demand. In fact, the Linux kernel itself uses mmap internally when it creates a new address space, as we saw in Section 4.3.2. When a user process or the kernel makes an mmap request, Linux calls upon the address-space management subsystem to create a new vm_area_struct descriptor for the requested memory region.

With this knowledge on Linux's memory management, it is fairly straightforward to take the KID field into consideration. First, the low-level paging subsystem is modified to recognize the KID field for PTE when an SP^3 page is being mapped. Thus, page-fault handlers for no-page and copy-on-write faults are modified to include a few lines of code that properly handles the KID field. The KID value to be used in the PTE is obtained from the vm_area_struct descriptor for the fault address. Therefore, the vm_area_struct structure is extended to include the KID value as an additional attribute. The merge/split logic of the address-space managing subsystem is also modified to take this KID value into account. The top-level mmap system call is modified to accept an additional argument, whose value is essentially a handle to the KID value to be used in the extended vm_area_struct descriptor. To achieve a consistent user-level interface, we added another layer of indirection when KID is referred to in the user space, hence the use of a handle in the `mmap` system call instead of using the actual KID value.

4.4.2 Modifications on signal and program entry

For the operating system to function correctly, it must understand the constraints implicitly imposed by SP^3. One is that the operating system cannot freely read from, or write to, the memory designated to the applications; it can do so, but the data it reads may be encrypted and thus unintelligible. Writing to the application's memory may corrupt the data since the memory may contain unintelligible 'random' data when the application reads the memory back. Another restriction is that the operating system cannot arbitrarily modify saved application contexts such as the program counter or the stack pointer (i.e, `EIP` and `ESP` in x86), because they are encrypted and stored in the secure domain context by means of the x86 extended exception frame.

With these constraints, the current implementation of the signal mechanism of Linux needs close attention. This is because the original Linux signal dispatcher intrusively writes into the user-space memory and also modifies the exception frame of the process. With SP^3, these activities will crash the application.

The original signal handler modifies the `EIP` value of a saved exception frame in order to deliver the signal: the new `EIP` value is the callback address of the signal handler function of the user process. However, in SP^3, the operating system should not directly specify the callback address by directly modifying the `EIP` of the saved exception frame. Thus, for the implementation of a user callback mechanism of signal (or other cases that require an overriding return-to-user address such as `fork`), the operating system must find ways other than directly overriding `EIP`.

We solved this problem by a scheme that utilizes a *user-level dispatcher*. When the user process first requests installation of a signal handler, the operating system saves the modified exception frame (Figure 3.2(b)) caused by the system call for later use. The operating system saves the signal handler's address and then returns to the user process with return value 0. When the operating system wants to signal the process later, it uses the saved exception frame but at this time, the return value contains the handler's address. Then, the user-level dispatcher, located between the system call interface and the C library, will test the return value and take appropriate actions.

Even with the user-level dispatcher, there still is a problem with the original signal implementation of Linux. When a signal is delivered, the operating system needs to save the current execution state so that it can resume the process after handling the signal. In Linux, this state information is saved on the user stack of the process, which is problematic in SP^3 since the user-level stack is encrypted. Also, the stack pointer is advanced to account for the saved state information, and this requires directly modifying the `ESP` field of the saved exception frame.

We solved this problem by allocating a special kernel memory designated for saving the state information. Also, we made the user-level signal handler use a separate memory for the stack. This way, the address of the signal handler's stack can be predefined, and therefore the `ESP` field need not be overridden by the kernel.

Linux's signal system is one of the few places where its programming interface is significantly changed: the user program is now responsible for signal dispatch, and signals cannot be nested due to the use of a separate stack for signals. However, these differences are not a problem for typical user applications since the user-level dispatcher and all of the start setup requirements can be handled by the C library. Also, few programs rely on nested execution of the signal handler.

4.5 Application programming

From the perspective of application programmers, SP^3 introduces a new programming environment. Using the protection interface of SP^3, privacy protection can be guaranteed. Application programming in SP^3 raises interesting issues, several of which are discussed next.

4.5.1 Application programming with SP^3

Although it is technically trivial to create an SP^3 executable as demonstrated in Section 4.3.1, transforming a non-SP^3 program to an SP^3 program usually requires a programmer to take a hard look at the information flow of the program. This is because what SP^3 provides is the programming environment and tools for securing selected segments of an application's code and data. In other words, SP^3 does not magically prevent incorrectly written programs with security holes from leaking secret information. In this context, summarized below are the things a programmer must consider when he writes an application program with SP^3.

First, it is the application programmer's responsibility to logically classify which data should be kept private and which data should be made public. For example, strings used as the `printf` argument are mostly public data, whereas variables holding sensitive information are private. Systematic solutions for information-flow analysis [114] can help the programmer detect potential problems. Also, this logical data classification should be realized by physical classification in the executable: private data should reside in the memory region mapped with KID, and public data should be located in the memory region with KID 0. SP^3 provides a C language extension based on annotation to facilitate this. Some data elements can be located statically at compile time, but in other cases, the program may need to move data elements around, depending on the runtime classification of the sensitivity of the data elements.

Second, although SP^3 provides an interface for a user application to protect its own code and data, it is entirely the application programmer's responsibility to write correct code that does not reveal sensitive information. Programmers can benefit from static/dynamic checker utilities [26] and integrity verification schemes [77,91] to further enhance security.

Last, programmers must be aware that the initial configuration at the program entry is very different from the traditional Application Binary Interface specification. Also, the semantics of the signal-delivery mechanism differ significantly. However, in most cases, a C library customized for SP^3 applications, such as our modified dietlibc [35], can mask most of these singularities. For example, the C entry code shown in Appendix B.1 is currently a part of the modified C library. This entry code, along with helper C functions, interprets the modified ABIs and implements the user-level signal dispatcher on behalf of user applications. To facilitate application development and packaging, we have developed a toolchain for creating an SP^3 application. The toolchain consists of encryption utilities, script files for compiling and linking, and the customized C library.

4.6 Summary and discussion

In Section 4.2, a close examination of the hello-world program in an SP^3 environment brought out the fundamental issues in both application programming and operating system construction. One fundamental aspect of programming in SP^3 is that a programmer must specify the information security category of each data item and write the application according to the specification.

This situation is analogous to multi-thread programming: running a single-thread program in a multi-threaded operating system does not make the program multi-threaded. The program has to be changed in order to fully utilize the multi-threaded environment. In the same respect, what SP^3 offers is the programming environment and APIs for building a secure application capable of running in an untrusted operating system. SP^3 does not magically protect an application that is insecure in itself.

In Section 4.3, we tackled the important subject of constructing an SP^3 executable from the source code. For the generation of executable SP^3 programs, we defined special object file sections in ELF standard to separate encrypted and unencrypted data, as depicted in Figure 4.5. By annotating the C source code, programmers can specify the object section to which data should belong, thus avoiding use of a special compiler. We avoided modification of compiler and linker by utilizing features of existing tools. This section also detailed how the SP^3 executable is loaded in a real untrusted operating system.

A new domain is created when exec is performed on an SP^3 executable, which contains encrypted sections. Thus, the ELF loader is modified to correctly load the encrypted sections of the executable file. The Alloc operation is performed at this point. The loader then performs mmap with appropriate KID values for the text, data, and stack memory regions. Program arguments, environment variables and ABI specific data are stored in the separate region instead of the default stack, since these data are not sensitive information.

In Section 4.4, we modified Linux, making it run correctly in the SP^3 environment. The KID integration in memory management routines only required an orthogonal extension of the virtual memory routines of Linux. At the lowest level of the virtual memory routines is the page-fault hander. We modified the page-fault hander to recognize and differentiate KID field of PTE structure. At the next level, Linux manages the virtual memory regions allocated to the user address space by means of a data structure known as vm_area_struct. Each vm_area_struct describes a memory region with the same property in the user's virtual address. We extended this structure to include KID for the region property. At the top of the virtual memory routines, Linux utilizes mmap for the general handling of mapping of memory to virtual address spaces. We modified the interface of mmap to accept KID value as an additional parameter.

Adapting Linux to the constraints imposed by SP^3 was a relatively easy task because Linux, like other operating systems, is indifferent to the contents of the user memory most of the time. Only a few parts, such as the signal delivery mechanism and program binary loader, meddle with user stack memory, and thus need to be modified. Considering the depth of changes made by SP^3 — we modified the semantics of paged memory and interrupt — the impact to the operating system can be considered very small. This is because of the orthogonality of the SP^3 paging semantics: the existing operating system's memory management routines only have to add the KID field into their internal structure and treat the field as another attribute of the memory.

In Section 4.5, we discussed the issues in application design and programming from the standpoint of an application developer who wants to benefit from SP^3 protection. The bottom line is that the application is ultimately responsible for writing correct code according to the program's own secrecy requirements because SP^3 does not protect privacy of an application that is written incorrectly.

CHAPTER V

MULTIMEDIA CONTENT PROTECTION

In the previous two chapters, we confined our discussion to the SP^3 system itself; in Chapter III, we detailed the hypervisor-based implementation of SP^3 protection model, followed by Chapter IV, in which we discussed the impact of the underlying SP^3 on the operating system and application programming. Since we covered the required details of the SP^3 system, a discussion on actual use cases of SP^3 seems to be in order.

In this chapter, we turn our attention to the topic of how to utilize the SP^3 protection to solve real world problems. As SP^3 is capable of removing the operating system from the trust base of the end user applications, SP^3 can solve many practical cases in which difficulties have been encountered because of an untrusted operating system or a malicious user who has a system privilege.

Among many problems that can be effectively solved by SP^3, here we choose the most compelling use case of SP^3: the protection of copyrighted media contents. Aside from the revenue lost from piracy, the many previous failed attempts to enforce copyrights, and the public perception that piracy is ineradicable, there are also technical merits for taking contents protection as an example; we show how SP^3 can secure the codec and media player, the exact vulnerability point from which many previous attempts for contents protection have suffered; we show how SP^3 can easily secure the media contents in both encoded and decoded forms; we show how SP^3 can guarantee complete secrecy of media contents from a contents provider to a playback hardware device equipped with a decryption circuitry.

5.1 Introduction

Enforcing protection against creating pirated copies from copyrighted media contents in a digital computer is a hard problem. History seems to suggest that copy-protection in computers is inherently futile, especially in systems where end-users can be entitled to be super users. In such systems, users with system privilege are allowed to replace kernel, load arbitrary kernel modules, and debug arbitrary processes. This allows a modestly skilled user with the malicious intent of making a pirated copy to disable, dismantle, bypass, or reverse-engineer any copy-protection measures relatively easily. Unfortunately, consumer PCs, which comprise the major target platforms of media contents playback, allow end-users to be super users and therefore vulnerable to digital piracy.

Making this situation hard to resolve is the fact that digital media contents, such as music and movies, are usually delivered in compressed and encoded form, thereby necessitating a software component for decoding the contents to produce raw digital signal. Since this decoder software deals with the media content (it produces the final decoded output at least), and is relatively easy to break if you have the super user privilege, this software becomes the single weakest point that can be easily exploited to subvert or bypass any copyright protection measures. In fact, many current and past copy-protection schemes have been subverted by exploiting this phase of multimedia processing. For a notable instance, the reported leakage of encryption keys for Advanced Access Content System (AACS) [1] used for Blu-ray and HD-DVD is believed to be obtained from reverse-engineering licensed media player software for PC [14].

Throughout the chain of procedures protected media contents have to go through to be finally played in an output device, the media player software presents itself as the security "hole" that is hard to secure. In contrast, storing and delivering protected contents seems relatively safe and secure: a protected content, in its encoded form, is usually encrypted so that it cannot be decoded properly without the right encryption key; a decoded raw digital content can be encrypted and then transmitted to a secure output device capable of decrypting the stream. For instance, the High-bandwidth Digital Content Protection (HDCP) standard [4] secures the digital contents transmitted between the display output port and the end display device. However, the media player software, which has to decrypt and decode the source media, and then to encrypt the raw data to deliver to a secure output device, must know all the relevant keys as well as the decrypted media content itself.

In this chapter, we demonstrate how to utilize SP^3 to secure the media player software, the most vulnerable point in the enforcement of media contents protection. Due to the SP^3 protection, an operating system and end users are prohibited from accessing the plaintext versions of both encoded and decoded media contents. Also prohibited is to reverse-engineer the media player itself because SP^3 essentially encrypts the media player instance all the time, including the code and data while it is being executed.

We also demonstrate the flexibility of the secure memory mapping facility introduced by the SP^3 user-level programming interface, which enables performing encryption and decryption without actually making calls to cryptography libraries. The encryption and decryption is done transparently by the underlying SP^3 system, which is just fulfilling the secure paging semantics of SP^3. The capability of loading multiple different symmetric keys is also confirmed to be useful in our media player because separate encryption keys are used for encoded source content and decoded raw content.

We designed and implemented a media player that can decode and playback a rights-protected digital audio file on a secure audio device that has an encryption-based secure channel. We chose the Advance Audio Coding (AAC) format — a digital audio compression standard — as the base codec for our audio player implementation. As an input, our audio player accepts an encrypted AAC audio file. The player decodes the AAC file and then sends out the raw output - in the Pulse-Code Modulation (PCM) format - to an audio device that understands the encrypted PCM audio stream. Even though a device driver for the audio device is necessary in order to manage the device, the device driver is outside of the trust base because the driver does not have the decryption key to decrypt the PCM audio stream.

The following sections are organized as follows. In Section 5.2, we present background information in the fields of digital media and copy-protection measures. In Section 5.3, our goals and assumptions are stated. In Section 5.4, we illustrate our design and implementation of a secure media player using SP^3 protection. In Section 5.5, we evaluate our media player implementation. We conclude in Section 5.6.

5.2 Background

In this section, we first introduce how multimedia contents, such as video and audio, are created, delivered and played in the digital computer system. Then we discuss methodologies for enforcing copyrights for such media contents.

5.2.1 Multimedia codec, container format, and media player

Digital multimedia data is usually stored and delivered in compressed form, due to the sheer volume of original raw digitized samples obtained from a source analog input. Exploiting the insensitivity of human perception to minute details, lossy-compression schemes, which intentionally discard insignificant information, are typically employed for multimedia compression as opposed to lossless compression schemes, which preserve every bit. This can produce highly compressed encoded media contents, the quality of which is sufficiently good when reproduced.

A *media codec* is a collection of software components that implements the algorithm for media compression and decompression. Codecs usually vary depending on the type of media (e.g., sound, still image, or video) and the industry standards the algorithms have to conform to. The component that implements a compression algorithm is called *coder* or *encoder*, and encoding refers to the action of applying the compression algorithm against raw media samples to produce compressed, encoded media contents. Conversely, the *decoder* decompresses media contents encoded by the matching encoder. The actual algorithms employ various digital signal processing methods, such as Discrete Cosine Transform (DCT), but most of the time what matters to end users is not the underlying algorithms but the standards that a media codec suite supports.

A *media container format* refers to a file specification by which a compilation of encoded media contents is collected. For example, a movie file consists of at least one encoded video stream plus multiple channels of audio streams. The container format binds these streams together and contains metadata that specifies the type of codec used, timing and compression parameters, etc.

A *media player* is a software frontend that is responsible for actually transferring decoded raw samples to output devices. When a multimedia file is opened, the media player interprets metadata from the container and selects appropriate decoders for the file. The media player then performs actual I/O operations for the decoded raw media samples to the output devices such as display device and sound card. In most cases, the media player simply makes system calls for I/O operations, and device drivers of the operating system perform the final step of transferring the raw media samples to output devices.

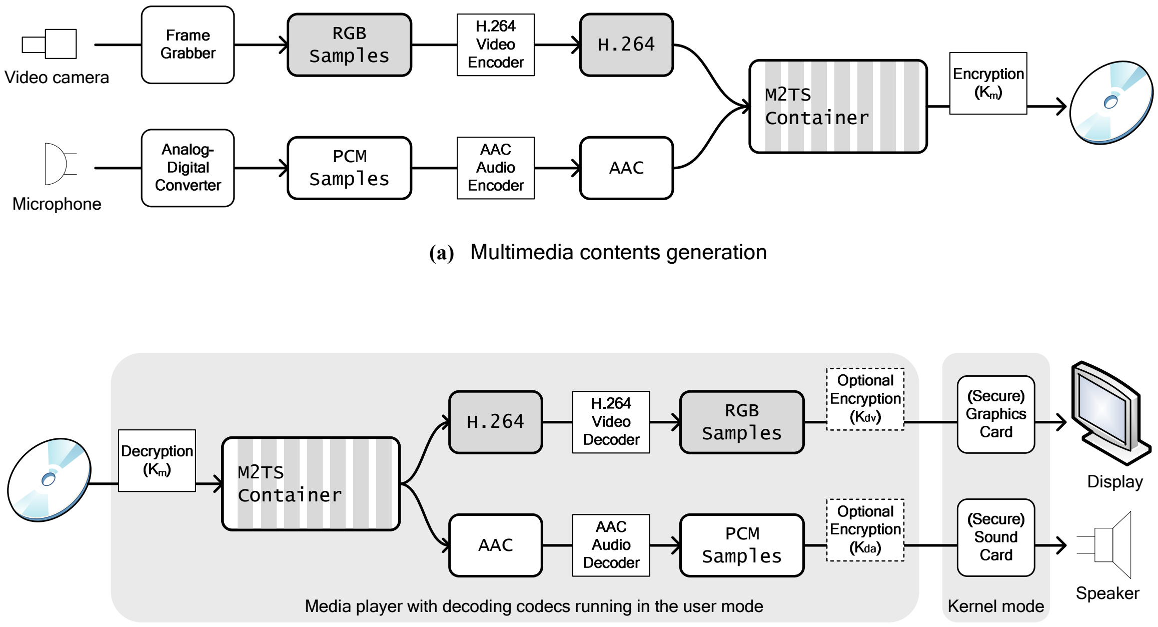
It should be noted that output devices can only accept raw media samples. Although the format of raw samples varies among devices, most of the devices work in the same way. For instance, to display a video frame, it is required to update the frame buffer of the display device pixel by pixel. For a sound device, a buffer containing raw Pulse-Coded Modulation (PCM) sound samples is required to play a sound.

5.2.2 Media copyright protection

Digital Rights Management (DRM) refers to the measures for preventing unauthorized copies of copyrighted digital contents from being made as well as ensuring the contents to be used only by the licensed persons or devices. DRM is usually applied to protect copyrighted media contents such as music and movies, and many DRM standards have been developed and accepted by the media and computer industry. For example, the DVD specification contains provisions for DRM called Content Scramble System (CSS), whose cryptography was broken [2], as well as Content Protection for Prerecorded Media (CPPM), which is not broken cryptographically, but can be easily circumvented. For Blu-ray and HD-DVD discs, a specification called Advanced Access Content System (AACS) [1] is used for DRM and content distribution. The High-bandwidth Digital Content Protection (HDCP) standard [4] enforces copyright across display port and display device by preventing an unlicensed display device from properly rendering the delivered contents. DRM may include digital-watermarks to trace the source medium of a pirated copy.

Clearly, cryptography plays a critical role in implementing DRM. For performance, symmetric-key block cipher or stream cipher is used to encrypt the media contents. The encryption key is then secured by a higher level of key management scheme, which varies platform by platform, depending on application specific requirements such as content distribution, device authentication, and key revocation scenarios. Sometimes, a media player also encrypts the decoded output stream — the raw samples — with the key of a licensed output device, so that only the licensed device can produce output correctly.

Figure 5.1 illustrates the steps taken when multimedia contents are generated and then played in the machine of an end-user. Figure 5.1 (a) shows how a content provider generates distributable media contents from the raw samples by encoding, packaging and encrypting the contents. In the figure, H.264 refers to the video encoding standard of MPEG-4, and AAC refers to one of the audio encoding standards. M2TS is a container file format standard. The end result of the content generation is the copy-protected contents encrypted with the symmetric key K_m. Figure 5.1 (b) shows how the delivered contents are reproduced in the machine of an end-user. A media player in the machine first decrypts and interprets the container file. The media player then calls upon the appropriate decoder to obtain raw samples acceptable for output devices. Then, the media player makes system calls to deliver



(a) Multimedia contents generation

(b) Multimedia contents playback

Figure 5.1: Steps for multimedia contents generation and playback. Shown in (a) is the sequence of sampling, encoding, packaging and encrypting of the source media from a content provider. The deliverable is therefore encoded and copy-protected. The playback sequence of the delivered contents is shown in (b). As can be seen, a media player running in the user mode performs the task of decryption and decoding.

the raw samples to device drivers, which run in the kernel mode. Note that there could be optional encryptions performed on the raw samples: this is intended to prevent raw data from being stolen and to allow playback only to the licensed devices. In the figure, symmetric keys K_{dv} and K_{da} are used for the secure graphics card and the secure sound card respectively.

Possible attacks against DRM measures can be classified into two categories: one category is direct cryptanalysis against cryptography and key management protocol, and the other one is the exploitation of vulnerable points in actual system implementations. Whereas the first category of attacks can indeed break a DRM measure [33], as in the case of DVD's CSS [2], most common and serious breaches occur due to the second category of attacks. For instance, it is extremely easy for an attacker to capture and copy the decoded output by intercepting system calls made from the media player, which amounts to effectively bypassing the entire copy-protection scheme. With some effort, reverse-engineering media players can even extract important keying materials whose leakage has a significant impact. In the much publicized instance of the leakage of encryption keys for Blu-ray's AACS, the keys are believed to be extracted from one of the licensed media player software.

Known as the 'trusted client' problem, the requirement of a trusted client program that performs the sensitive task of decryption and decoding is responsible for a great deal of previous breaches of DRM measures. This is because in many situations the trusted client program can be easily hacked and analyzed by an attacker who has a system privilege. Unfortunately, this is the situation for most of the consumer PC environments where end-users are capable of doing whatever they want to do.

In this chapter, we directly address this trusted client problem by utilizing the protection provided by SP^3. A media player, including a decoder, is protected by SP^3 in such a way that the internal workings of the media player can never be revealed to the outside, including the operating system. The media player takes encoded media in encrypted form, and produces decoded raw samples also in encrypted form.

5.3 System goals and assumptions

Our first goal is to secure the media contents and media player from untrusted end-users by means of the secrecy protection and the secure-memory mapping facility provided by SP^3. We say the security is broken when the attacker can obtain plaintext of either encoded or decoded media contents. Obviously, any leakage of the keying materials constitutes a security compromise.

Our second goal is to allow correct playback of media contents only to the licensed device. To this end, we assume each device has its own cryptography key which is unique and whose value is never known to the end user. The key is only known to the device manufacturer and the media content provider who licenses the device. The device may have a unique device identifier for the end user to use, who may supply the identifier to the content provider for a device authorization.

We assume the following parties/components to be trusted from the perspective of media content providers: hardware (i.e., computer, graphics/sound cards, and end display/sound device), client programs (i.e., media codecs and media player), and underlying SP^3 system.

We assume the following to be untrusted and potentially malicious from the perspective of media content providers: operating system, end users, operating system administrators, and device drivers.

Note that the above assumptions are by far the most realistic assumptions that suitably reflect the concerns of media content providers, except for the SP^3 system, which we introduce in this work. In fact, without SP^3, the assumptions lead us exactly to the trusted client problem from which many breaches have originated. In this respect, SP^3 is the price that has to be paid to solve the problem.

The above assumptions mostly follow the situation of consumer PC platforms, which are relatively ‘open’ as opposed to ‘closed’ platforms such as stand-alone DVD or Blu-ray players, digital set-top boxes, or digital TVs. For closed platforms, it can be reasonable to assume that the software installed within the hardware boxes — such as operating system and device drivers — is trusted, since it is hard for end-users to gain access to the pre-installed software. However, the situation for the majority of consumer PCs and handheld devices does not resemble that of closed platforms. It should be also emphasized that our assumptions make the problem harder, not easier.

Promoting practicality, we also restrict ourselves to the ‘generic’ versions of media codec and cryptography primitives; we use standard-conforming generic codecs such as H.264, AAC, or MP3, as well as standard cryptography such as AES. By this, we mean we do not attempt to achieve security by obscurity of our tools. Note that this is a restricting assumption that makes the problem harder to solve, but makes a solution more practical.

Finally, we only deal with symmetric key cryptography in this work by assuming that key-management issues are fairly orthogonal problems that can be dealt with separately. This assumption is reasonable because most existing DRM schemes exhibit this separation.

5.4 Design and implementation

In this section, we present the design and implementation of a media player that is protected by SP^3, thereby achieving the stated goals of securing the copyrighted media contents from untrusted end users and limiting the contents playback only to the licensed device. In the following discussion, we only focus on audio media contents. However, the generality of our approach is not lost because video contents can be protected in the same way.

5.4.1 Design

We secure the media player and codec by making them an SP^3 program encrypted with a symmetric key K_p (subscript ‘p’ denotes ‘player’) chosen by the content provider.

This implies that the content provider, or a trusted partner of the content provider, needs to supply the media player in SP^3 executable format. This is an obvious requirement since our trust assumption prevents the use of a media player from an untrusted third party. When executed in the SP^3 environment, the complete running instance of the media player is protected, thereby preventing spying or reverse-engineering.

We deliver the end-user a copy-protected audio file encrypted with a symmetric key K_m ('m' denotes 'media'), which is also chosen by the content provider. When the media player opens the audio file, the media player has to decrypt the file with K_m. The media player could just call decryption functions, and to do so is safe because the media player is protected, and therefore embedding K_m in the media player itself is safe. However, the SP^3 programming interface provides a convenient way of accessing a file with its extended `mmap()` semantics in which a program can specify a KID value to be used for the memory mapping. We choose `mmap()` for accessing the media player file.

As stated in the previous section, the sound device of the end-user is a licensed device and therefore associated with a device-unique symmetric key K_{da} ('da' denotes 'device-audio'). This key is only known to the device manufacture and the content provider. The kernel driver for this device does not know the key. This sound device is capable of generating sound from both normal Pulse-Code Modulation (PCM) samples and encrypted PCM samples. The former ability allows the sound device to work for normal, copyright-free sounds, and the latter allows it to work for rights-enforced sounds. Since only the device manufacturer and the content provider know the encryption key K_{da}, which is unique to the device, capturing encrypted PCM samples from one machine and playing them on another machine is prohibited.

The media player has to encrypt the decoded PCM samples using the symmetric key K_{da}, and we can use the same `mmap()`-based strategy for encryption. When we create a memory buffer for storing decoded PCM samples, we can allocate the buffer by calling the `mmap()` system call with `MAP_ANONYMOUS`, specifying a KID value at the same time.

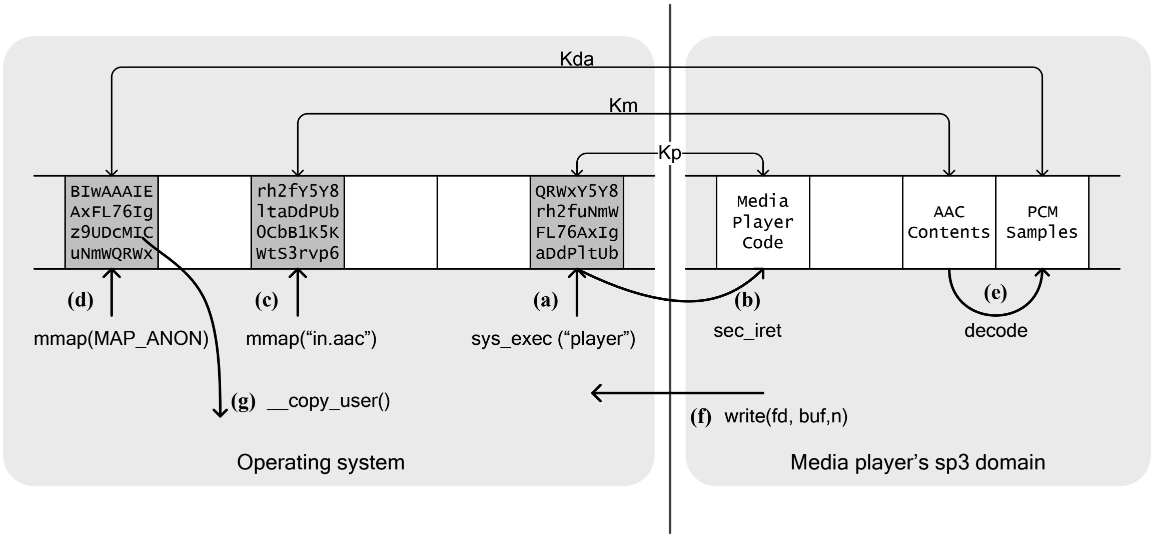


Figure 5.2: Memory contents of a hypervisor-based SP^3 system when a secure media player is being executed. Each encrypted memory page from the operating system side (left) has its decrypted counterpart in the private area for the SP^3 domain of the media player (right). These page pairs are associated with symmetric keys. Steps (a) to (g) indicate the system events caused by the execution of the media player.

This way, the decoder just places plaintext PCM samples in the buffer, and then the media player passes the buffer pointer to the audio device driver. When the device driver, running in the kernel mode, accesses the buffer, the buffer contains encrypted PCM samples due to the SP^3 secure paging semantics. Even if the device is using DMA, the device sees PCM samples encrypted with K_{da}.

The SP^3 metadata section of the media player executable securely carries all the symmetric keys, namely K_p, K_m, and K_{da}, as detailed in the previous chapters. Using SP^3's PKI, these keys are securely delivered to the underlying SP^3 system when the `Alloc` instruction is called. This instruction is executed when the SP^3 media player program is initially launched. This instruction, as defined in the previous chapters, creates a new SP^3 domain for the media player, loads all three symmetric keys to the SP^3's key database, and initializes the corresponding bits in the permission bitmap. After the `Alloc` instruction, the media player runs within the newly-allocated context of the SP^3 domain.

Figure 5.2 illustrates the memory pages and the interactions among them as the media player is being executed in an SP^3 system. Here we assume the hypervisor-based SP^3 implementation, which is detailed in Chapter III. Hence, the hypervisor keeps the decrypted copy of a sp3-protected page in the private memory region. On the left, the figure shows the memory pages and their contents as seen by the operating system. On the right are the memory pages accessible only to the hypervisor and the media player running within its SP^3 domain context. On top, the figure shows the connections between encrypted and decrypted images, along with associated symmetric keys. As can be seen, the operating system is allowed to see only the encrypted images.

Figure 5.2 also shows a sequence of system events caused by the execution of the media player. First, when the sp3 executable is executed in step (a), the operating system's loader maps the encrypted executable binary to the memory. The operating system then schedules the media player process: by executing `sec_iret` in step (b), the current SP^3 domain of the system switches to the SP^3 domain of the process, enabling the process to execute the decrypted media player code. Later, in step (c), the media player opens the source audio file using `mmap()`, which causes the operating system to load the verbatim — thus encrypted — image of the source audio file. In step (d), the media player allocates a buffer by calling anonymous `mmap()`, which causes the operating system to allocate memory pages. When the media player process addresses the mapped regions, the process accesses the pages for decrypted images. This is because the underlying hypervisor redirects page tables when the SP^3 domain is entered. The media player decodes the source audio and put the result in the buffer, as shown in step (e). In step (f), the media player performs an I/O by making a `write()` system call, passing the PCM buffer pointer to the kernel driver. In step (g), the device driver of the kernel handles the `write()` system call, ultimately accessing the PCM buffer, but the buffer contains encrypted PCM data because the driver is running in the kernel mode.

5.4.2 Implementation

Since SP^3 currently supports only Linux and an SP^3 C library based on dietlibc, we implemented our secure media player in Linux with the SP^3 programming facilities detailed in Chapter IV.

For our audio codec, we used the Advance Audio Coding (AAC) implementation of Freeware Advanced Audio Code codec suite [3], known as faac and faad2, version 1.28 and 2.7 respectively. The decoder, faad2, is compiled and linked statically with the front-end media player program. To link with the SP^3 C library, which lacks floating-point math functions, the use of the math library is avoided by replacing the floating point operations with fixed point operation, which the faad2 decoder already supports. The decoder library went through the required process of verification and modification for adapting to the SP^3 programming environment.

For our media player implementation, we modified and extended a decoder frontend program included in the faad2 codec suite. The original frontend program only decodes an input file and drops a file as the output. We therefore added the function of actually playing the sound by writing to our secure sound device, which will be described shortly.

The file I/O of the media player frontend also went through significant changes, including an implementation of C file I/O based on the `mmap()` interface. The buffer for the decoded PCM samples is also implemented using the `mmap()` interface. These `mmap()` calls enabled us to completely avoid calling any of the cryptography routines at all. All that was necessary was to specify an SP^3-specific parameter in one of the parameters of the `mmap()` system call in order to identify which KID is used for the memory mapping.

Although we do not have an actual secure sound card that can accept encrypted PCM samples and produce sound, we developed a device driver for an imaginary secure sound device. Our Linux device driver is a character device implementing the `write()` device callback. The driver's `write()` is invoked when a user program performs `write()` system call on a device file associated with the device driver. This is the standard way in Linux a user program plays sound. The analogous `read()` device callback is left unimplemented because we do not record sound in this work. A device file for our sound driver is created at `/dev/secsound`, to which our media player writes decoded PCM samples. Our device driver only accepts a PCM buffer aligned on 4KiB boundary. This is due to the granularity of the block encryption scheme used to implement SP^3 secure paging.

It should be emphasized that this device driver does not require K_{da}, meaning that the content provider does not have to trust the device driver. Our device driver implementation, however, does include K_{da} and AES decryption routines because we want to test and verify the whole scenario in the absence of the actual secure sound device. The decryption routines in the current driver implementation should be discarded with a real secure device.

5.5 Evaluation

Due to the nature of this work, our evaluation questions are mostly qualitative:

- Does our implementation achieve the stated security goals?
- How difficult is it to develop a media player as an SP^3 application?
- What are the sources of overhead other than those directly caused by SP^3?
- What are the limitations of our media player implementation?

5.5.1 Security

Our implementation achieves the stated security goals for the following reasons. First, the entire execution of the media player and codec is cryptographically shielded by SP^3. This means that unless an attacker breaks the cryptography, it is impossible for the attacker to reverse-engineer the program in an attempt to extract intermediate media contents. Second, the source media contents are delivered to the media player as encrypted, and the decrypted contents are only visible to the media player. In our implementation, the actual decryption is done transparently by the underlying SP^3, which we trust. Therefore, at no point are the decrypted source media contents left vulnerable to potential spying from the untrusted parties such as end user and operating system. Third, the decoded output samples are also encrypted before leaving the media player. As with the source contents case, the underlying SP^3 performs the actual encryption. Therefore, the decoded output samples are also prevented from being stolen. Last, the three encrypted entities —the media player, source media contents, and output samples — are secured by encryption keys known only to the content provider and to the licensed hardware. This implies that even if a user unlawfully acquires all three encrypted entities, it is impossible for him to correctly reproduce the contents unless he owns the device that the media player and contents are licensed to.

5.5.2 Ease of development

Although it took several days for the author to understand the Linux sound device and the faac/faad2 source code sufficiently enough to perform modifications, it took about 45 minutes to actually scramble sources and make required changes to the faad2 core codec. The required changes were mostly for adapting to the SP^3 C library and cleaning up unnecessary code that caused compilation and linkage errors. However, the modifications made to the frontend took two days to finish. This is because of the time spent implementing the playback function and the `mmap()`-based file I/O, as well as testing and debugging in the SP^3 environment. For a person who is already familiar with the source code of a codec and a media player, it is reasonable to say that converting the program to the SP^3 environment is an easy task that can be done in a short amount of time.

5.5.3 Overhead

The use of the fixed-point operations instead of the floating-point operations can impact decoder performance. We measured the difference in decoder performance in the native Linux environment, since the SP^3 version of C library does not support floating point math routines. This is the reason why fixed-point operation was used in our implementation. We prepared two versions of decoder: one uses fixed-point operation and the other uses floating point operation. We measured the time taken for each version to decode the same input file. We performed the experiment on a machine with Intel Core 2 2.13 GHZ dual core processor, using only single thread, measuring wall-clock time.

The result shows that it took 90% more time for the fixed-point decoder to finish than for the floating-point decoder. However, the fixed point decoder is still efficient enough to handle AAC files in real-time. For the fixed-point decoder, it took an average of 9.7 seconds to decode a 619-second-long, AAC-encoded piece of classical music. Given that the reason for adopting fixed-point operation is a technical incompatibility that can be easily fixed by providing an SP^3 version of the C math library, we can say that the 90% performance overhead is not inherent.

5.5.4 Limitations

Our current secure media player implementation has limitations. One is that our definition of the secure output device is too simplistic. Only one symmetric key is associated with the secure sound device. It cannot handle the situation, for example, where the device has to be licensed by multiple content providers who mutually distrust, and therefore cannot simply share the symmetric key. Although our assumption of a single symmetric key simplified our discussion, a sophisticated device key management scheme will be necessary to reflect complex trust relationships of real world scenarios. For example, we can use a PKI scheme that can dynamically load/revoke multiple symmetric keys in the device.

Another limitation is that the encryptions performed against encoded media contents and output raw samples must follow the same block-cipher parameters used by the underlying SP^3 system. This is because the media player has chosen the option to utilize `mmap()` for transparent encryptions. Therefore, should there be any need to avoid the use of SP^3's block-cipher, the media player always has the option to just use their own cryptography library. Alternatively, it is also possible for the SP^3 to support different cipher modes that an application can choose.

Finally, end-users may be wary of the SP^3 layer. Obviously, the SP^3 is a system layer that is off-limits to end-users who might physically own the machine, and users might not like the idea. However, we argue that SP^3 is the minimal price to pay to circumvent the trusted client problem. Other solutions to this problem tend to limit the rights of the end users much more severely, since these solutions try to make the end-users' system a 'closed' system that is effectively owned by rights holders, rather than the end users. In this respect, we also argue that SP^3 provides the most flexible and reasonable solution that can sufficiently satisfy the needs of both end-users and rights-holders. This is because SP^3 preserves the right of an end user to perform as a super user, including the power to replace operating system kernel components. At the same time, rights-holders can distribute their copyrighted contents safely and securely.

5.6 Conclusion

In this chapter, we demonstrated the strength and effectiveness of SP^3 protection by illustrating an important use case of SP^3: the protection of copyrighted media contents. We showed how SP^3 can secure the codec and media player, the exact vulnerability point from which many previous attempts for contents protection have suffered. It is demonstrated that SP^3 can easily secure the media contents in both encoded and decoded forms. The end result is the complete secrecy of media contents from a contents provider to a playback hardware device.

To demonstrate the concept, we designed, implemented and evaluated a secure media player that can decode and play a rights-protected digital audio file on a secure audio device that has an encryption-based secure channel. We chose the Advance Audio Coding (AAC) format as the base codec for our audio player implementation. As an input, our audio player accepts an encrypted AAC audio file. The player decodes the AAC file and then sends out the raw output — in the Pulse-Code Modulation (PCM) format — to an audio device that understands the encrypted PCM audio stream. Even though a device driver for the audio device is necessary in order to manage the device, the device driver is outside of the trust base because the driver does not have the decryption key to decrypt the PCM audio stream.

We conclude this chapter by emphasizing that SP^3 can serve as a very attractive DRM infrastructure element in which end-users' rights are preserved. With SP^3, users can still install any software of their choice, perform any administrative jobs, and even replace operating system kernels.

CHAPTER VI

CONCLUSION

This thesis demonstrates the feasibility of separating information protection from resource management in systems software. Among the common security properties of secrecy, integrity, and availability, protection for secrecy and integrity can be efficiently separated out from the responsibilities of the traditional operating system. The operating system is then only responsible for availability, which is the *raison d'etre* for an operating system. This separation greatly reduces the size and complexity of the trusted part for information protection, resulting in a more secure system that can tolerate a compromise in the operating system.

To this end, I designed and implemented a system architecture called Software-Privacy Preserving Platform (SP^3) that can efficiently separate the roles of protection and management. Defined in Chapter II, SP^3 realizes the separation by augmenting the way that software accesses memory by including a mechanism for cryptography-based information protection. SP^3 effectively provides information protection directly to user-level applications. As the unit of protection, we chose a memory page, which is also the unit of resource management by the operating system. This choice enables a minimally intrusive introduction of a security layer at the deepest level of the software stack.

The practicality of the SP^3 system is demonstrated in Chapter III by implementing SP^3 using a hypervisor. The new memory access semantics were emulated efficiently by utilizing the capability of a hypervisor. The SP^3 system was implemented on the Xen hypervisor and the Linux operating system.

Further explored in Chapter IV are challenges and solutions in the new programming environment that SP^3 introduces. In this environment, the operating system must be aware of the limited capability imposed by SP^3; the operating system is no longer capable of obtaining meaningful information from protected memory pages. Also, an application programmer must decide on a rule governing which data should be shared with the operating system. We provide application programming support for users to adapt to this new programming environment.

In Chapter V, we turned our attention to how to utilize SP^3 to solve real world problems. We chose a compelling use case of SP^3: the protection of copyrighted media contents. We designed and implemented a media player that can protect rights-enforced multimedia contents. We showed how SP^3 can secure the codec and media player, the weakest points of rights management infrastructure.

This thesis studied a system construct that has a potential to solve many system security problems. This system construct can be considered as a security framework upon which many practical solutions can be constructed. In the following sections, we explore the unique potential of the construct introduced in this thesis and review opportunities for further investigation. In Section 6.1, we highlight the advantages of our solution in comparison with related systems. In Section 6.2, we outline the integrity protection support in the SP^3 system. In Section 6.3, we conclude with future work.

6.1 Advantages of SP^3

The advantages can be demonstrated by comparing our solution with three related approaches: trust-decoupling based on virtual machine isolation, platform integrity protection based on the trusted computing, and other untrusted operating system solutions such as Overshadow.

First, let us consider the isolation based approach. Using a hypervisor, one can easily obtain a virtualized hardware instance completely isolated from other virtual machines running in parallel. In the isolated virtual machine, one can install ‘trusted’ software separated and protected from an untrusted operating system. Flicker [71], for example, is a system that aims to provide complete isolation. Proxos [101] goes a step further and allows applications to choose a different operating system for different system calls so that the applications can selectively use the trusted operating system for ‘sensitive’ services.

However, although isolation-based solutions can technically sidestep untrusted operating systems, the usefulness of these solutions is still substantially small. This is because these solutions fail to recognize the fact that many practical end applications actually need to run within the full context of an untrusted operating system. For instance, the secure media player in Chapter V requires a full-fledged operating system complete with virtual memory, file systems, process scheduler, driver model, etc. But, in order to implement the secure media player with isolation-based solutions, one has to either supply his own ‘trusted’ copy of a full-fledged operating system, or develop a stand-alone media player that can execute without an operating system. The former approach is contradicting the original reason why we need to remove the operating system from the trust base in the first place. The latter approach requires extensive source code modification and probably results in a very inflexible application that does not coexist well with others because we are forgoing the benefits of using an operating system. Selectively choosing operating systems for different system calls does not help because the desire for a full-fledged operating system does not go away, let alone the difficulty of figuring out how to divide the trust base using the irregular, coarse, whimsically designed system call interface.

On the other hand, the solution presented in this thesis differs from isolation-based solutions, benefitting a large number of practical applications. This is because our approach allows applications to execute within the full context of the untrusted operating system, meaning that the applications can fully utilize the operating system’s services without the fear of losing sensitive data. The secure media player presented in Chapter V highlights the seamless integration with an untrusted operating system while fulfilling the security goals.

Second, let us compare our solution with trusted computing. The trusted computing standard uses a trusted piece of hardware called the Trusted Platform Module (TPM), which can securely calculate integrity hashes — called measurement — on hardware and software during the boot process. If the measured values are different from the values expected from a trust authority, the system is presumed to be altered in an unauthorized way and the system is deemed to be unsuitable for executing sensitive operations from the perspective of the trust authority. This way, one can guarantee the genuineness of the system before taking the next step. It should be noted that TPM does not actually prevent unauthorized alterations; if the system is altered, only the components that depend on the genuineness of the system will not proceed.

However, although it is technically allowed to alter the software in a system with a TPM, the price of doing so by end-users would be their incapability of executing any software from the trust authority. In order to use the software, one must not violate the genuineness requirement. This is a serious problem because the genuineness requirement can be very stringent and restrictive to end-users in most practical situations, usually depriving the end-users of any privilege and limiting what end-users can do to the mere action of just executing a set of programs. For example, the trusted computing version of the secure media player can be implemented in such a way that the hardware, BIOS, operating system kernel, device drivers, and other dependent components are measured and verified, and the system should also be configured in such a way that nobody can become super users, perform debugging, redirect output of the media player, etc. Only then, the trusted computing version of the secure media player will execute.

On the other hand, our system does not limit end-users' capabilities: end-users are allowed to do practically anything, including kernel replacement. As demonstrated by our secure media player implementation, we can please both end-users and content providers by giving a high degree of freedom to end-users and protecting the copyrighted media contents at the same time. We believe that our system is by far the most reasonable solution that resolves the conflicting interests between end-users and content providers.

Last, let us consider Overshadow [28]. Since Overshadow shares its motivations and goals with us, it can be said that the above advantages of our approach over virtual machine isolation and trusted computing also apply to Overshadow. Overshadow also attempts to provide protection in an unmodified[1] guest operating system to unmodified user applications. Therefore, Overshadow does not have in-guest support for user applications explicitly written under the assumption of an untrusted operating system.

On the other hand, our system is designed to provide explicit support for applications written under the untrusted operating system assumption. Instead of hiding the underlying protection mechanism, we try to expose it so that applications can make the most of it. The extension of `mmap()` system call and its use in the secure media player implementation proves the advantages of our system.

6.2 Integrity protection implementation outline

In this section, we outline an implementation of integrity protection system with a hypervisor-based SP^3 system.

6.2.1 Background

The Keyed-Hash Message Authentication Code (HMAC) is used to authenticate and verify the contents of a page. HMAC is a standard for generating a Message Authentication Code (MAC) using a cryptographic hash function in combination with a secret key. A MAC is a small piece of information attached with a message. The recipient of the message calculates his own MAC from the message and compare his own MAC with the

[1] Although whether patching a binary constitutes a modification or not is a matter of opinion, the use of the term ‘unmodified’ in Overshadow is understood to mean that source code is not modified.

one attached with the message. If it is different, the recipient of the message knows that the message is not authentic. HMAC uses a secret key for calculating a MAC. Therefore, HMAC has the property that without knowing the secret key, a valid MAC cannot be generated from the message. It also has the property that prevents an attacker from retrieving the secret key by analyzing a valid pair of a message and a MAC.

We assume that the vendor of trusted application programs and the vendor of hypervisor trust each other; hence they know and share the secret key that is used to generate valid MAC code. The other parties, such as operating system, end users, and operating system administrator, are not trusted. Therefore, the end users or system operator who installs the trusted application program do not have access to the secret key. This implies that an end user cannot create a valid message/MAC pair that is acceptable by the underlying hypervisor.

Our integrity protection system is classified into two categories: *protection registration* and *integrity enforcement*. Protection registration is the activity initiated by an application program requesting the hypervisor for protection of one of its pages. Integrity enforcement is the activity taken by the hypervisor to actually impose write-prevention to the registered pages.

Figure 6.1 illustrates the design of the system architecture for the protection registration and integrity enforcement procedures. Shown on the left is the protection registration procedure. In this procedure, the trusted application program makes the initial request by invoking a hypercall for registration requests. The hypervisor then verifies the request by computing HMAC code. Then the hypervisor updates its internal registry that keeps track of all the integrity protected pages, and then returns to the user.

Shown on the right of Figure 6.1 is the integrity enforcement procedure. In this procedure, the entry to the hypervisor is caused by the page fault induced by a write attempt made in the guest. The page fault handler of the hypervisor consults the registry to determine whether the fault is generated by the actual integrity protection violation. If so,

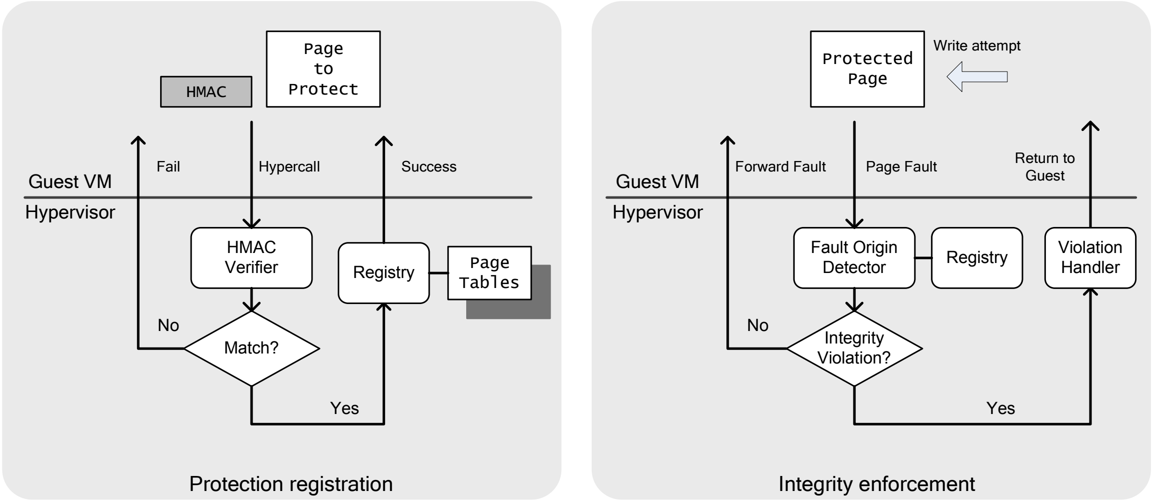


Figure 6.1: System architecture for protection registration and integrity enforcement. Shown on the left is the protection registration procedure and on the right the integrity enforcement procedure.

the hypervisor handles the violation and takes countermeasure. If not, the page fault is forwarded to guest virtual machine for the guest operating system to handle.

6.2.2 Protection registration

During the protection registration phase, the authentication between the hypervisor and the application takes place. This authentication is required to verify that the application is the real one that we trust. There are many alternatives in doing this, but we chose keyed-Hash Message Authentication Code (HMAC), whose characteristics are discussed in the previous section.

In our design, the hypervisor and the developer of the trusted application program share the secret key. Using the secret key, the developer generates the valid MAC for a page that he wants to protect using the integrity protection of the hypervisor. When the page is being registered, the MAC is also supplied to the hypervisor along with the page. The hypervisor then performs its own HMAC on the page to verify the validity of the MAC that was supplied. If the MAC is valid, hypervisor then proceeds and registers the page to the

SP^3 internal registry. Note that the secret key is never revealed: the program carries not the secret key but just a valid MAC generated by the key.

The registration request should be a direct request to the hypervisor. Therefore, registration is a hypercall made by the application. This hypercall takes the address to a page, the MAC code, and the type of protection as parameters.

6.2.3 Integrity enforcement

The hypervisor enforces the security rules on the registered pages to protect them. Once registered, the page is protected by the hypervisor. The security rules are associated with the protection type that was specified when the page was registered.

The enforcement of security rules consists of violation detection and countermeasure. The hypervisor detects violation by manipulating the page table entries that maps the registered page, which is to trigger an event when there is a violation to the security rule. When an event is triggered due to a violation, the hypervisor perform countermeasure to handle this violation. The types of countermeasures can also be specified when the page is registered.

To provide maximum flexibility to the operating system, the policies are enforced as advisory rather than mandatory: the operating system is free to violate the policy, but should there be any violation, the operating system will get an unexpected result from the operation that violated the policy. For example, writing to an integrity-protected page will not prevent the operating system from proceeding, but the write operation will not be reflected on the page. Also, the hypervisor is always informed of any violation.

Allowing hypervisor to take actions on misbehaving guest can lead to nasty behavior in the guest. This is because it amounts to breaking the system rule assumed by the guest operating system. Although our threat model does not include protecting guest operating system, we need to try to minimize potential negative impact on the guest world, even if it is violating the protection rules.

There are two types of countermeasure that can be specified. They are *report-only* and *fault-inject*. With report-only, the violation will only be notified to the hypervisor, and the write operation is reflected on the page. This actually does not prevent write operating from happening, but it is the least intrusive countermeasure which is not breaking guest operating system rules. With fault-inject, the violation will be treated like a page fault caused by writing on read-only page, producing a page fault in the guest. Note that it might confuse guest operating system because the guest operating system will get a page fault from a perfectly good condition.

Note that we could also introduce a third type of countermeasure: for the *write-squash* type, write operation will be silently ignored. The semantics of write-squash is that the program that caused the violation will proceed, without the expected result of previous write operation. However, the actual technical method to achieve this countermeasure could be quite involved, requiring disassemble of the current instruction that caused the violation. This could be technically challenging, especially x86 systems. Therefore, we do not consider the write-squash type.

6.3 Future work

Future work can be further pursued in the following directions.

- Standalone SP^3 implementation: The size and complexity of the trust base can be substantially lowered if SP^3 is implemented as a standalone system layer. Running only one guest operating system, the layer can also utilize the hardware virtualization features introduced in recent processors to further reduce the size.

- Untrusted network stack: End-to-end network communication can be secured transparently by utilizing underlying SP^3 encryption. However, implementing this would require non-trivial network stack modification because the network packets vary in size and headers are prepended frequently, which makes it hard to work with the page-granular SP^3 encryptions.

- More SP^3 use case scenarios: SP^3 can help solve many security problems when the operating system or system administrators cannot be trusted. For example, SP^3 can secure a web browser and its private data in a public shared computer. Distributed computing projects based on donated computing resources over the Internet can utilize SP^3 to ensure the integrity of the computation results.

- Performance improvement: The performance of the hypervisor-based SP^3 implementation can be improved by better algorithms such as using page directory entries in addition to page table entries for invalidation of pages.

APPENDICES

APPENDIX A

Extension to x86 System Architecture

A.1 New instructions

Alloc, Free, and secure iret instruction as defined in Section 2.4.1 are embodied in the form of new instructions added to the x86 instruction set. The encoding of these instructions makes use of the reserved area of instruction encoding space. Given below serves as the definition of these instructions in the form of GNU C in-line assembly code, which can be used to generate the instruction encoding.

```
/* p  [Ed]: pointer to {Ks}Kp+  (via EAX as REG field)
 * kids[Gd]: pointer to array of kids  (via EBX as R/M field)
 * returns new sid via [Gd]  */
static inline unsigned sec_alloc(void* p, unsigned* kids) {
    unsigned sid;
    asm volatile (
        ".word 0x3e0f \n"                      /* op code */
        ".byte 0xd8 \n"                        /* modRM */
        "nop \n"
        "mov %%ebx, %0 \n"
        : "=r"(sid)
        : "a"(p), "b"(kids)
    );
    return sid;
}
```

```
static inline void sec_free(unsigned sid) {
    asm volatile (
        ".word  0x3f0f \n"                      /* op code */
        ".byte  0xc3 \n"                        /* modRM */
        "nop"
        :
        : "b"(sid)
    );
}

/* 2 byte opcode.. */
static inline void sec_iret(void) {
    asm volatile (
        ".word  0x3a0f \n"                      /* op code */
        ".align 16, 0x90 \n"
        : : :"memory"
    );
}
```

A.2 Secure exception frame

If an SP^3 domain is running when a processor gets interrupted, a special 128-byte memory block is written to the kernel stack, followed by the standard x86 exception frame. The memory block contains the state of the interrupted program in an encrypted form. When it is decrypted, each field of the block is defined as follows.

```
/* spa-x86 loader header definition */
typedef struct spa_secframe {
    unsigned xsid[4];           /* 128-bit extended sid */
    unsigned blk_gs_lo;         /* reserved */
    unsigned blk_gs_hi;         /* reserved */
    unsigned blk_fs_lo;         /* reserved */
    unsigned blk_fs_hi;         /* reserved */
    unsigned blk_es_lo;         /* reserved */
    unsigned blk_es_hi;         /* reserved */
    unsigned blk_ds_lo;         /* reserved */
    unsigned blk_ds_hi;         /* reserved */
    unsigned blk_ss_lo;         /* reserved */
    unsigned blk_ss_hi;         /* reserved */
    unsigned blk_cs_lo;         /* reserved */
    unsigned blk_cs_hi;         /* reserved (byte offset 64) */
    unsigned etype;             /* exception type */
    unsigned extrainfo;         /* reserved */
    unsigned __st0_offset72;    /* unallocated */
    unsigned edi;               /* saved edi register */
    unsigned esi;               /* saved esi register */
    unsigned ebp;               /* saved ebp register */
    unsigned __st0_offset88;    /* unallocated */
    unsigned ebx;               /* saved ebx register */
    unsigned edx;               /* saved edx register */
    unsigned ecx;               /* saved ecx register */
    unsigned eax;               /* saved eax register */
    unsigned eip;               /* saved eip register */
    unsigned cs;                /* saved cs  register */
    unsigned eflags;            /* saved eflags register */
    unsigned esp;               /* saved esp register */
    unsigned ss;                /* saved ss  register */
} spa_secframe_t;
```

`xsid` stores an hashed SID value of the interrupted program. The hashing algorithm uses a pseudo-random seed unique to the current SID instance in order to avoid forgery.

The `etype` field is set to SYSCALL when the source of the interrupt is software-generated. In this case, the registers contain system call arguments, therefore they are not cleared to zero upon interrupt. When returning from the interrupt, if the field is SYSCALL, then SP^3 does not recover the registers from the saved registers with the exception of `esp`, `eip`, `eflags` `cs`, and `ss`.

For all other interrupt/exceptions, the `etype` field is set to a different value other than SYSCALL. In this case, the registers are cleared. When returning from the interrupt, then SP^3 recovers the registers from the saved registers.

APPENDIX B

Modified System V ABI

B.1 CRT main entry point

Shown below is the SP^3 counterpart of the C's _start program entry. The initial entry point is shared by all signal handler callbacks in our modified ABI. The code thus shows how to invoke user-level signal dispatcher.

```
/* for initial entry, %ebx is 0 and %ebp is addr to stack
   for signal, %ebx is signal number, %esi is handler */
_sp3_start:
    cmpl     $0x0, %esp
    jnz      1f                 /* not an sp3 program */
    movl     %ebp, %esp
    xorl     %ebp, %ebp
    cmpl     $0x0, %ebx
    jnz      3f                 /* signal dispatcher */
    push     %esi               /* arguments */
    jmp      2f
1:  xorl     %ebp, %ebp
    push     %esp
2:  call     __sp3__clear_bss   /* extern C function */
    call     __sp3__cstart      /* extern C function */
    push     %eax
    call     _exit
3:  push     %eax               /* signal parameter */
    push     %ebx               /* signal number */
    push     %esi               /* handler address */
    call     __sp3__dispatch_signal /* extern C fn */
    popl     %eax               /* sigreturn sequence */
    movl     $119, %eax         /* __NR_sigreturn */
    int      $0x80
```

B.2 GNU linker script

GNU linker (ld) supports Linker Command Language, which is a script language describing how to collect sections from object files, how to specify address offsets for segments, and how to package the binary into an executable file. Shown below is the ld script for generating an SP^3 application, which conforms to the ELF executable format. The script arranges ELF segments in such a way that each segment always has page-aligned start and end addresses. The resulting executable file is post-processed to encrypt segments that has to be encrypted.

```
OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_spa_start)
SEARCH_DIR("/usr/local/lib");
SEARCH_DIR("/lib");
SEARCH_DIR("/usr/lib");
PHDRS
{
    headers    PT_PHDR FILEHDR PHDRS;
    elfmetaro  PT_LOAD FILEHDR PHDRS;/* elf metadata, plaintext */
    spameta    PT_LOAD FLAGS (6);    /* spa metadata, plaintext */
    spaunsecro PT_LOAD FLAGS (4);    /* readonly sections, plaintext */
    spaunsecrw PT_LOAD;              /* writable sections, plaintext */
    spasecro   PT_LOAD;              /* text/rodata sections, encrypted*/
    spasecrw   PT_LOAD;              /* data/bss sections, encrypted */
    gnustack 0x6474E551 FLAGS (6);
}
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = 0x08048000); . = 0x08048000 + SIZEOF_HEADERS;
  .interp         : { *(.interp) } :elfmetaro
  .hash           : { *(.hash) } :elfmetaro
  . = ALIGN(0x1000);
  .spa.allocinfo    : { *(.spa.allocinfo) } :spameta
  .spa.pages        :
  {
    *(.spa.page.0)
    *(.spa.page.1)
    *(.spa.page.2)
    *(.spa.page.3)
    *(.spa.page.4)
    *(.spa.page.5)
  } :spameta
  . = ALIGN(0x1000);
  .spa.rodata.unsec   : { *(.spa.rodata.unsec) } :spaunsecro
  . = ALIGN(0x1000);
  .spa.data.unsec   : { *(.spa.data.unsec) } :spaunsecrw
  . = ALIGN(0x1000);
  .init           : { KEEP (*(.init)) } :spasecro =0x90909090
  .text           :
  {
    *(.text .stub .text.* .gnu.linkonce.t.*)
    KEEP (*(.text.*personality*))
    *(.gnu.warning)
  } :spasecro =0x90909090
  PROVIDE (__etext = .);
  PROVIDE (_etext = .);
  PROVIDE (etext = .);
  .rodata         : { *(.rodata .rodata.* .gnu.linkonce.r.*) } :spasecro
  .rodata1        : { *(.rodata1) } :spasecro
  .eh_frame_hdr   : { *(.eh_frame_hdr) } :spasecro
  .eh_frame       : ONLY_IF_RO { KEEP (*(.eh_frame)) } :spasecro
  . = ALIGN(0x1000);
  . = DATA_SEGMENT_ALIGN (0x1000, 0x1000);
  .eh_frame       : ONLY_IF_RW { KEEP (*(.eh_frame)) } :spasecrw
  .got            : { *(.got) } :spasecrw
  . = DATA_SEGMENT_RELRO_END (12, .);
  .got.plt        : { *(.got.plt) }
  .data           :
  {
    *(.data .data.* .gnu.linkonce.d.*)
    KEEP (*(.gnu.linkonce.d.*personality*))
    SORT(CONSTRUCTORS)
  } :spasecrw
  .data1          : { *(.data1) } :spasecrw
  _edata = .;
  PROVIDE (edata = .);
  .data_filler :
  {
    __spa__data_filler_start = .;
    . = (ALIGN(0x1000) - .) ? ( . + ( ALIGN(0x1000)- .) - 4 ) : (. + 0xffc );
    LONG(0xDA7AF111);
  } :spasecrw
  __spa__data_filler_end = .;
  . = ALIGN(0x1000);
  __bss_start = .;
  .bss            :
  {
   *(.dynbss)
   *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
```

```
    . = ALIGN(32 / 8);
  } :spasecrw
  . = ALIGN(32 / 8);
  _end = .;
  PROVIDE (end = .);
  . = DATA_SEGMENT_END (.);
}
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] AACS - Advanced Access Content System. http://www.aacsla.com/specifications/.

[2] DeCSS. http://www.cs.cmu.edu/~dst/DeCSS/.

[3] Freeware Advanced Audio Code. http://www.audiocoding.com/.

[4] HDCP - High-bandwidth Digital Content Protection. http://www.digital-cp.com/.

[5] System V Application Binary Interface, Intel386 architecture processor supplement, 4th ed.

[6] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *Proceedings of the 4th ACM Conference on Computer and Communications Security (CCS)*, pages 36–47, Zurich, Switzerland, April 1997.

[7] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer 1986 Technical Conference*, pages 93–113, Atlanta, GA, June 1986.

[8] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, San Jose, CA, October 2006.

[9] Andrew W. Appel. Deobfuscation is in NP, 2002.

[10] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Vancouver, Canada, June 2000.

[11] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 1–18, Santa Barbara, CA, August 2001.

[12] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Rolf Neugebauer Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, Bolton Landing, NY, October 2003.

[14] BBC news. Hi-def DVD security is bypassed. `http://news.bbc.co.uk/2/hi/technology/6301301.stm`.

[15] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41–46, Anaheim, CA, April 2005.

[16] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th USENIX Security Symposium*, pages 305–320, Vancouver, Canada, August 2006.

[17] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 267–284, Copper Mountain, CO, December 1995.

[18] Matt Blaze. A cryptographic file system for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security (CCS)*, pages 9–16, Fairfax, VA, November 1993.

[19] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[20] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, Austin, TX, December 2001.

[21] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 1st International Symposium on Code Generation and Optimization (CGO)*, pages 265–275, San Francisco, CA, March 2003.

[22] Derek Bruening and Vladimir Kiriansky. Process-shared and persistent code caches. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE)*, pages 61–70, Seattle, WA, March 2008.

[23] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, pages 57–72, San Diego, CA, August 2004.

[24] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[25] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–156, Saint Malo, France, October 1997.

[26] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, Alexandria, VA, November 2006.

[27] Benjie Chen and Robert Morris. Certifying program execution with secure processors. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS)*, Lihue, HI, May 2003.

[28] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, Seattle, WA, March 2008.

[29] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 140–153, Charleston, SC, December 1999.

[30] Christian S. Collberg and Clark D. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *Transactions on Software Engineering*, 28(8):735–746, 2002.

[31] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL)*, pages 184–196, San Diego, CA, January 1998.

[32] Landon P. Cox and Peter M. Chen. Pocket hypervisors: Opportunities and challenges. In *Proceedings of the 8th IEEE Workshop on Mobile Computing Systems and Applications (HotMobile)*, Tucson, AZ, February 2007.

[33] Scott Crosby, Ian Goldberg, Robert Johnson, Dawn Song, and David Wagner. A cryptanalysis of the high-bandwidth digital content protection system. In *Proceedings of the 2001 ACM Workshop on Digital Rights Management*, Philadelphia, PA, November 2001.

[34] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 161–174, San Francisco, CA, April 2008.

[35] DietLibC. diet libc – a libc optimized for small size. `http://www.fefe.de/dietlibc/`.

[36] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 17–30, Brighton, United Kingdom, October 2005.

[37] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, CO, December 1995.

[38] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, 2003.

[39] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–151, Seattle, WA, October 1996.

[40] Jason Franklin, Vern Paxson, Adrian Perrig, and Stefan Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 375–388, Alexandria, VA, October 2007.

[41] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Reconstructing I/O. Technical report, Computer Laboratory, University of Cambridge, August 2004.

[42] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS)*, Boston, MA, October 2004.

[43] Tal Garfinkel, Ben Pfaf, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine–based platform for trusted computing. In *Proceedings of the*

19th ACM Symposium on Operating Systems Principles (SOSP), pages 193–206, Bolton Landing, NY, October 2003.

[44] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed System Security (NDSS)*, pages 191–206, San Diego, CA, February 2003.

[45] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 5th USENIX Security Symposium*, San Jose, CA, July 1996.

[46] Robert P. Goldberg. *Architectural Principles for Virtual Computer System.* PhD thesis, Harvard University, Cambridge, MA, 1972.

[47] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, pages 34–45, June 1974.

[48] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[49] Hermann Härtig. Security architectures revisited. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, pages 16–23, Saint–Emilion, France, September 2002.

[50] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 66–77, Saint Malo, France, October 1997.

[51] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components – small kernels versus virtual-machine monitors. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, page 22, Leuven, Belgium, September 2004.

[52] Hai Huang, Padmanabhan Pillai, and Kang G. Shin. Design and implementation of power-aware virtual memory. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 57–70, San Antonio, TX, June 2003.

[53] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[54] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual.*

[55] Intel Corporation. Intel trusted execution technology. `http://www.intel.com/technology/security/`.

[56] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3.* Intel, 2008.

[57] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, pages 59–74, Washington, D.C., August 2003.

[58] Trevor Jim, Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[59] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 91–104, Brighton, United Kingdom, October 2005.

[60] Bernhard Kauer. OSLO: Improving the security of trusted computing. In *Proceedings of the 16th USENIX Security Symposium*, pages 229–237, Boston, MA, August 2007.

[61] Taeho Kgil, Laura Falk, and Trevor Mudge. ChipLock: Support for secure microarchitectures. In *Proceedings of the Workshop on Architectural Support for Security and Anti-virus (WASSA)*, pages 130–139, Boston, MA, October 2004.

[62] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 223–236, Bolton Landing, NY, October 2003.

[63] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, San Francisco, CA, August 2002.

[64] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 161–176, Seattle, WA, November 2006.

[65] Butler W. Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton, NJ, March 1971.

[66] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 178–192, Bolton Landing, NY, October 2003.

[67] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 169–177, Cambridge, MA, November 2000.

[68] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–250, Copper Mountain, CO, December 1995.

[69] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 27–31, Washington D.C., October 2003.

[70] Lionel Litty and David Lie. Manitou: A layer-below approach to fighting malware. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 6–11, San Jose, CA, October 2006.

[71] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM European Conference on Computer Systems (EuroSys)*, pages 315–328, Glasgow, Scotland, April 2008.

[72] Microsoft Corporation. NGSCB: Next-generation secure computing base. `http://www.microsoft.com/resources/ngscb`.

[73] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC)*, pages 175–181, Washington, DC, December 1997.

[74] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. A crawler-based study of spyware on the web. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2006.

[75] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–142, Saint Malo, France, October 1997.

[76] National Institute of Standards and Technology. Advanced Encryption Standard (AES), (FIPS–197), 2001.

[77] Taejoon Park and Kang G. Shin. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 4(3):297–309, May/June 2005.

[78] Nick L. Petroni, Jr and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 103–115, Alexandria, VA, October 2007.

[79] Birgit Pfitzmann, James Riordan, Christian Stüble, Michael Waidner, and Arnd Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research Division, Zurich Laboratory, April 2001.

[80] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[81] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *Proceedings of the 16th USENIX Security Symposium*, pages 275–290, Boston, MA, August 2007.

[82] Dan R. K. Ports and Tal Garfinkel. Towards application security on untrusted operating systems. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Security (HotSec)*, San Jose, CA, July 2008.

[83] Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: using an annotation toolkit to build DoS-resistant software. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[84] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

[85] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the SimOS machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[86] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[87] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[88] Love H. Seawright and Richard A. MacKinnon. VM/370 – a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[89] R. Sekar, V. N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. Duvarney. Model–carrying code: A practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–28, Bolton Landing, NY, October 2003.

[90] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 335–350, Stevenson, WA, October 2007.

[91] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–16, Brighton, United Kingdom, October 2005.

[92] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, Washington, DC, October 2004.

[93] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185, Charleston, SC, December 1999.

[94] Jonathan S. Shapiro, John Vanderburgh, Eric Northup, and David Chizmadia. Design of the EROS trusted window system. In *Proceedings of the 13th USENIX Security Symposium*, pages 165–178, San Diego, CA, August 2004.

[95] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 166–176, Berkeley, CA, May 2000.

[96] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of the 1st ACM European Conference on Computer Systems (EuroSys)*, pages 161–174, Leuven, Belgium, April 2006.

[97] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstations hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 1–14, Boston, MA, June 2001.

[98] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, pages 160–171, San Francisco, CA, June 2003.

[99] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, pages 339–350, San Diego, CA, December 2003.

[100] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Science of Computer Programming*, 62(2):122–144, October 2006.

[101] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating system configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 279–292, Seattle, WA, November 2006.

[102] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[103] Trusted Computing Group. Trusted platform module (tpm) specifications. `https://www.trustedcomputinggroup.org/specs/TPM/`.

[104] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *Proceedings of the 16th USENIX Security Symposium*, pages 239–256, Boston, MA, August 2007.

[105] Chang-Hao Tsai and Kang G. Shin. DPS: A distributed proportional-share scheduler in computing clusters. In *Proceedings of the 2nd IEEE International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID)*, Munich, Germany, May 2007.

[106] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 148–162, Brighton, United Kingdom, October 2005.

[107] Carl A. Waldspurger. Memory resource management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–194, Boston, MA, December 2002.

[108] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[109] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isonation for linux using mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 31–44, Brighton, United Kingdom, October 2005.

[110] Jisoo Yang and Kang G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE)*, pages 71–80, Seattle, WA, March 2008.

[111] Bennet S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, May 1994.

[112] Bennet S. Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of the 1st USENIX Workshop on Electronic Commerce*, pages 155–170, New York, NY, July 1995.

[113] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, 1998.

[114] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the*

18th ACM Symposium on Operating Systems Principles (SOSP), pages 1–14, Banff, Canada, October 2001.

[115] Xin Zhao, Kevin Borders, and Atul Prakash. Towards protecting sensitive files in a compromised system. In *Proceedings of the 3rd International IEEE Security in Storage Workshop*, San Francisco, CA, December 2005.