A Study of Security Isolation Techniques

RUI SHU, PEIPEI WANG, SIGMUND A. GORSKI III, BENJAMIN ANDOW, ADWAIT NADKARNI, LUKE DESHOTELS, JASON GIONTA, WILLIAM ENCK, and XIAOHUI GU,
North Carolina State University

Security isolation is a foundation of computing systems that enables resilience to different forms of attacks. This article seeks to understand existing security isolation techniques by systematically classifying different approaches and analyzing their properties. We provide a hierarchical classification structure for grouping different security isolation techniques. At the top level, we consider two principal aspects: mechanism and policy. Each aspect is broken down into salient dimensions that describe key properties. We break the mechanism into two dimensions, enforcement location and isolation granularity, and break the policy aspect down into three dimensions: policy generation, policy configurability, and policy lifetime. We apply our classification to a set of representative articles that cover a breadth of security isolation techniques and discuss tradeoffs among different design choices and limitations of existing approaches.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Security and privacy** → **Systems security;**

Additional Key Words and Phrases: Security isolation, access control, resilient architectures

ACM Reference Format:
Rui Shu, Peipei Wang, Sigmund A. Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. 2016. A study of security isolation techniques. ACM Comput. Surv. 49, 3, Article 50 (October 2016), 37 pages.
DOI: http://dx.doi.org/10.1145/2988545

1. INTRODUCTION

Isolation is a foundational concept of multitasking computing systems that dates back to at least Multics [Corbató and Vyssotsky 1965]. Multics introduced the concept of a process. Each process has its own address space, which isolates memory between programs. This process abstraction and its memory isolation simplifies software development by providing a modular abstraction for programs. It also provides security isolation, which was a primary goal of Multics. The process abstraction also enables fault isolation. Security isolation and fault isolation are similar, but there are also differences between them. Fault isolation ensures that a fault in one partition does not affect others. Therefore, if a software program has a bug, it will not crash the entire system. However, security isolation means that even if the security of a partition is compromised, the adversary cannot breach the security of other partitions. Finally, the

This work was supported by the National Security Agency under the Science of Security Lablet at North Carolina State University.
Authors' addresses: R. Shu, P. Wang, S. A. Gorski III, B. Andow, A. Nadkarni, L. Deshotels, J. Gionta, W. Enck, and X. Gu, North Carolina State University, Department of Computer Science, 890 Oval Drive, Box 8206, Engineering Building II, Raleigh, NC 27695-8206; emails: {rshu, pwang7, sagorski, beandow, anadkarni, ladeshot, jjgionta, whenck, xgu}@ncsu.edu.

DOI: http://dx.doi.org/10.1145/2988545

process abstraction enables performance isolation. That is, a supervisor scheduler can ensure each process fairly shares the CPU compute time.

However, imperfection in security isolation may be exploited by several attacks. For example, in cloud computing, since many users share one physical computing platform, security threats may come from co-location of services [Ristenpart et al. 2009]. Shared resources such as CPU, network, or caches also make the construction of covert channel communication possible. Therefore, an efficient security isolation technique is required for security assurance.

This survey article focuses on studying security isolation, one of the building blocks for resilient architectures. Resilient architectures are among the key challenges in security research. For example, it is identified as one of the hard problems for the NSA's science of security research [Nicol et al. 2012]. Security isolation relates to the principles of *least privilege* and *privilege separation* [Saltzer and Schroeder 1975].

Security isolation is commonly used in two broad ways, depending on the threat model. First, it can be used to safely execute a program that is not trusted or not yet trusted. Second, it can be used to harden a system that must run trusted programs that have an increased attack surface. For example, it is beneficial to isolate network-facing daemons such as Web, Domain Name System (DNS), and Email servers. In this case, the software is trusted but assumed to contain vulnerabilities. The goal of isolation is to limit what can happen if a vulnerability is exploited. In contrast to isolating an untrusted program, techniques that isolate a trusted program can leverage support of the developer.

There are many different techniques and abstractions for achieving security isolation. Different techniques have different benefits and drawbacks. The goal of this survey article is to define a structure for classifying existing techniques to better understand how to build future systems. A secondary goal is to understand where future research can enhance existing isolation techniques. For example, more resilient architectures can be built by incorporating some of the more dynamic benefits of performance isolation into security isolation.

We provide a hierarchical classification structure for grouping existing security isolation techniques. The top level of the classification hierarchy includes two principal design aspects, which are *mechanism* and *policy*. These two broad concepts are commonly used to describe systems in literature and textbooks. For each design aspect, we define a set of dimensions that describe the key properties relevant to the design aspect. For example, we describe the mechanism aspect with respect to the security enforcement location and isolation granularity. Dimensions are then broken down into categories that cover different design choices of specific security isolation techniques. For example, the enforcement location might be in hardware or within a supervisor such as a kernel or hypervisor. We iteratively developed this classification hierarchy by studying a set of representative articles that propose a diverse set of security isolation techniques. We then characterize these works within the hierarchy.

To better understand different security isolation techniques, we provide an introduction to the tradeoffs among different design choices (i.e., categories) for each dimension (e.g., enforcement location, isolation granularity). We compare and contrast different categories within each dimension based on performance overhead, requirement of code for the isolated subject, and the security assurances provided by the technique. Because generalizations are often difficult to make, we discuss tradeoffs with respect to scenarios, providing concrete examples from the literature.

This article makes the following contributions:

—*We provide a classification hierarchy for describing existing security isolation techniques*. We use the hierarchy to classify a set of representative articles that cover all possible design choices for each design dimension.

—*We discuss the tradeoffs among different design choices.* In doing so, we help designers of future security isolation systems to best understand which security isolation techniques to incorporate.
—*We discuss the limitations of existing approaches.*

The remainder of this article proceeds as follows. Section 2 introduces our article selection process, hierarchical classification structure, and evaluation criteria. Section 3 describes different design choices for the security isolation mechanism. Section 4 describes classification with respect to policy. Section 5 concludes.

2. METHODOLOGY

This section describes our survey methodology. We begin by describing our article selection process. We then describe the classification hierarchy we devised to characterize the various security isolation techniques. This hierarchy was iteratively created as we investigated various techniques. Finally, we describe the criteria to evaluate the tradeoffs among different design choices for implementing security isolation.

2.1. Paper Selection

We surveyed a wide variety of security isolation techniques to cover various categories of approaches and identified relevant articles from the proceedings of the top venues for academic research in systems security (e.g., The USENIX Security Symposium, the IEEE Security & Privacy Symposium (S&P), the ACM Conference on Computer and Communications Security (CCS), and the Network and Distributed System Security Symposium (NDSS) and systems (e.g., the ACM Symposium on Operating Systems Principles (SOSP), the USENIX Symposium on Operating Systems Design and Implementation (OSDI) and the USENIX Annual Technical Conference (ATC)). For this study, we focus on more recent proceedings (within the past 5 to 10 years), as recent work has digested and applied many of the security isolation concepts that have been discovered in the past 50 years. We supplement our list of publications with well-known articles and systems that are frequently discussed in current literature. Some of these techniques (e.g., User-Mode Linux) may not have corresponding publications.

Our article selection process is integrated with the creation and refinement of the classification hierarchy described in Section 2.2. In doing so, we ensure that our sample provides representative examples of a wide range of security isolation techniques. As we explore different classifications, we note areas with limited coverage and seek to identify articles to fill deficits. Our goal is not to quantify the use of any specific security isolation characteristic but rather to cover a large range of prior work representing the various design choices involved in security isolation. Tables I and II list some of the articles discussed in this article, classified using the methodology we describe further in this section. At least two authors validated the classification for every article.

2.2. Hierarchical Classification Structure

We create a hierarchical classification structure to describe different facets of security isolation techniques, as shown in Figure 1. We give each level of the hierarchy a name. The top level consists of two *aspects*, which represent the two orthogonal design issues: mechanism and policy. The middle level consists of different design *dimensions* for each aspect. The bottom level (leaf nodes) comprises different *categories* representing the potential design choices for each dimension. To classify a particular security isolation technique, we characterize it along each dimension for both aspects (i.e., policy and mechanism). For each dimension, we place the technique into one of the categories that matches the design decision adopted by the technique.

In Tables I and II, we also highlight a set of representative security isolation techniques in order to present that these techniques are evaluated throughout all the

Table I. Examples of Relevant Work on Mechanism

Categories	Subcategories	Example papers
Enforcement Location		
Physical Host		IceFS [Lu et al. 2014]
Hardware Component		**Cerium** [Chen and Morris 2003], **Flicker** [McCune et al. 2008], **SICE** [Azab et al. 2011], Haven [Baumann et al. 2014]
Supervisor	*Hypervisor*	**Xen** [Barham et al. 2003], Terra [Garfinkel et al. 2003], Svgrid [Zhao et al. 2005], QEMU [Bellard 2005], sHype [Sailer et al. 2005], CHAOS [Chen et al. 2007], Overshadow [Chen et al. 2008], SP^3 [Yang and Shin 2008], TrustVisor [McCune et al. 2010], HUKO [Xiong et al. 2011], Appshield [Cheng et al. 2013], Proxos [Ta-Min et al. 2006]
	Library OS	Drawbridge [Porter et al. 2011], Bascule [Baumann et al. 2013], Mirage [Madhavapeddy et al. 2013], Graphene [Tsai et al. 2014]
	Container	MAPbox [Acharya and Raje 2000], Vx32 [Ford and Cox 2008], TxBox [Jana et al. 2011], ARMor [Zhao et al. 2011], MBOX [Kim and Zeldovich 2013], **Capsicum** [Watson et al. 2010], Pivot [Mickens 2014], **MiniBox** [Li et al. 2014], ARMlock [Zhou et al. 2014a], OpenVZ [Kolyshkin 2006], Linux-VServer [Soltesz et al. 2007], Linux Containers (LXC) [Helsley 2009], Docker [Merkel 2014], **PREC** [Ho et al. 2014]
Intra-Application	*Code Rewriting*	Software Fault Isolation (SFI) [Wahbe et al. 1994], MiSFIT [Small 1997], microkernels [Singaravelu et al. 2006], SFI for CISC [McCamant and Morrisett 2006]
	Compiler	Modula-3 [Cardelli et al. 1989], SPIN [Holzmann 1997], certifying compiler [Necula 1998], J-kernel [Von Eicken et al. 1999], TALx86 [Crary et al. 1999], language-based security [Schneider et al. 2001], Singularity [Hunt and Larus 2007], Native Client [Yee et al. 2009]
	System Loading	Aurasium [Xu et al. 2012]
Isolation Granularity		
Guest OS		**Xen** [Barham et al. 2003], User-mode Linux [Dike et al. 2001], OpenVZ [Kolyshkin 2006], Denali [Whitaker et al. 2002], Shuttle [Shan et al. 2012]
Application Group		MAPbox [Acharya and Raje 2000], LXC [Helsley 2009], Terra [Garfinkel et al. 2003], Pea-Pod [Potter et al. 2007], Jails [Kamp and Watson 2000]
Application		Inktag [Hofmann et al. 2013], Privexec [Onarlioglu et al. 2013], **Cerium** [Chen and Morris 2003] Overshadow [Chen et al. 2008], TxBox [Jana et al. 2011], MBOX [Kim and Zeldovich 2013], Polaris [Stiegler et al. 2006], **MiniBox** [Li et al. 2014], Svgrid [Zhao et al. 2005], Janus [Wagner 1999], AirBag [Wu et al. 2014], **SICE** [Azab et al. 2011]
Sub-application		Program shepherding [Kiriansky et al. 2002], **Capsicum** [Watson et al. 2010], **PREC** [Ho et al. 2014], **Flicker** [McCune et al. 2008], Native Client [Yee et al. 2009], TrustVisor [McCune et al. 2010], Embassies [Howell et al. 2013], SFI for CISC [McCamant and Morrisett 2006], SFI [Wahbe et al. 1994], Addroid [Pearce et al. 2012], AdSplit [Shekhar et al. 2012], Vx32 [Ford and Cox 2008], Gazelle [Wang et al. 2009], Codejail [Wu et al. 2012], Adjail [Ter Louw et al. 2010]

*Note that in the table we highlight a set of techniques that are fully classified in all dimensions.

Table II. Examples of Relevant Work on Policy

Categories	Example papers
Policy Generation	
Automatic	Systrace [Provos 2003]
Manual	MAPbox [Acharya and Raje 2000], **PREC** [Ho et al. 2014], **Cerium** [Chen and Morris 2003], **Xen** [Barham et al. 2003], Trustvisor [McCune et al. 2010], **Capsicum** [Watson et al. 2010], **MiniBox** [Li et al. 2014], **Flicker** [McCune et al. 2008], **SICE** [Azab et al. 2011]
Policy Configurability	
Reconfigurable	**Cerium** [Chen and Morris 2003], Singularity [Hunt and Larus 2007], **Xen** [Barham et al. 2003], **Flicker** [McCune et al. 2008], **Capsicum** [Watson et al. 2010], Inktag [Hofmann et al. 2013], **PREC** [Ho et al. 2014], **MiniBox** [Li et al. 2014]
Non-reconfigurable	Libra [Ammons et al. 2007], **SICE** [Azab et al. 2011]
Policy Lifetime	
Always On	SFI [Wahbe et al. 1994], Program shepherding [Kiriansky et al. 2002], **Xen** [Barham et al. 2003], **Cerium** [Chen and Morris 2003], Polaris [Stiegler et al. 2006], SFI for CISC [McCamant and Morrisett 2006], Singularity [Hunt and Larus 2007], Vx32 [Ford and Cox 2008], **Flicker** [McCune et al. 2008], Gazelle [Wang et al. 2009], **SICE** [Azab et al. 2011], Inktag [Hofmann et al. 2013], **MiniBox** [Li et al. 2014]
On Demand	**Capsicum** [Watson et al. 2010], TxBox [Jana et al. 2011], **PREC** [Ho et al. 2014]

*Note that in the table we highlight a set of techniques that are fully classified in all dimensions.

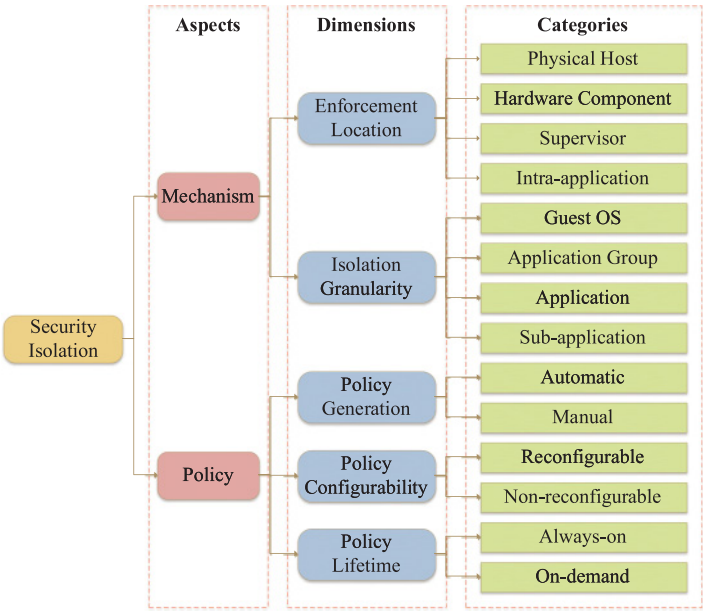


Fig. 1. Hierarchical classification structure for security isolation.

categories in all dimensions. However, due to the space constraints, it is difficult to cover all techniques. The examples we list in the tables are representative techniques that belong to each category.

Mechanism: The *mechanism* aspect describes *how* the security isolation goals are achieved. We break the mechanism into two dimensions: *enforcement location* and *isolation granularity*. Note that while some categories under the two dimensions may

appear similar, they capture semantically different design decisions, as will be clear further in this section.

The enforcement location defines where the mediation occurs, that is, the location of the conditional logic that allows or denies access to resources. The mediation logic must be non-circumventable and tamper-proof [Anderson 1972] to ensure security. The most common location for the mediation logic is within a supervisor of some form (e.g., kernel or hypervisor); however, some security isolation techniques leverage hardware enforcement or place mediation within an application itself (e.g., inline reference monitors). Finally, we include physical host separation for completeness.

In contrast to enforcement location, the isolation granularity describes the scope of the subject being isolated. The categories range from coarse to fine. At the coarsest granularity is the entire operating system. The next (finer) granularity is of a group of applications, followed by the granularity of a single application. Finally, the last granularity describes isolation among different parts of an application.

We separate the isolation granularity from the enforcement location, because isolation at a particular granularity can be enforced by reference monitors at various enforcement locations. For example, Flicker [McCune et al. 2008] uses hardware to isolate a sub-application subject called a Piece of Application Logic (PAL), while Capsicum [Watson et al. 2010] uses a supervisor to isolate parts of an application. Additionally, the isolation granularity alone (i.e., without considering the enforcement location) may provide a coarse estimate of the performance overhead and security guarantees of a particular technique. For instance, isolating different guest OSes using OS virtualization requires duplication of OS kernels, storage stacks, and memory management. In contrast, isolating groups of applications within the same OS (e.g., using Linux Containers [Helsley 2009]) avoids the duplication of core OS services, resulting in less resource consumption. At the same time, we can broadly assume that isolation of guest OSs is more robust from the security standpoint, as the isolated guests depend on a very small number of shared host services, although the specifics of how the isolation is implemented may challenge this assumption.

Policy: The *policy* aspect describes a set of rules that, when enforced, achieve the isolation goals. We break the policy aspect down into three dimensions: *policy generation*, *policy configurability*, and *policy lifetime*. Note that the categories do not represent types of policies but the characteristics of an isolation policy.

The policy generation dimension captures how the policy is created. Sometimes, policy generation is simple and can be done manually (e.g., isolate an entire guest OS). Other times, policy generation is more nuanced; for example, systrace [Provos 2003] must identify all of the system calls and resources that an application may need to access. In this case, automatic policy generation is more appropriate.

The policy configurability dimension captures whether or not the policy can change and, if so, how. Some techniques (e.g., OS virtualization) hard code the security isolation policy. Such policies cannot be changed without recompiling and deploying new software. Other techniques use a policy that is separate from the mechanism, which enables configurability. As we discuss later, the ability to configure a policy may be limited to specific points in time (e.g., on reboot) or may be changed on the fly.

Finally, the policy lifetime dimension captures the time at which the security isolation enforcement mechanism is activated. That is, there are cases wherein the policy may not be enforced throughout the system's lifetime but may only be activated on demand after the occurrence of a certain event. For example, in Capsicum [Watson et al. 2010], the execution of a code segment causes a part of the application to be isolated. The lifetime dimension helps us separate such cases where the policy is enforced on demand from the usual case where the policy is enforced all the time.

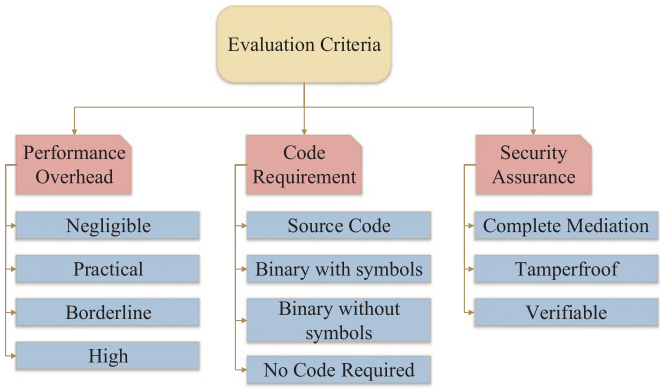


Fig. 2. Evaluation Criteria.

2.3. Evaluation Criteria

Our classification hierarchy defines the design space for security isolation techniques. For each dimension, the system designer must decide among different categories.[1] Therefore, our discussion of each dimension concludes with an evaluation of the broad tradeoffs between the categories.

Figure 2 shows the three areas of evaluation: *performance overhead*, *code requirement*, and *security assurance*. The first two criteria capture the practicality of incorporating the security isolation technique into a production system, whereas the third criterion describes the security guarantees that can be achieved.

Performance Overhead: The performance overhead describes the additional application execution time incurred due to the isolation mechanism. Note that this differs from resource overhead. For example, an isolation system may require additional disk resource (i.e., resource overhead) for storing the policy files while enforcing the isolation policies might incur extra execution time by requiring the system to intercept each disk access (i.e., performance overhead). Further, most security isolation articles that we studied do not describe the resource overhead. Thus, in this article, we only focus on the performance overhead.

Security isolation techniques proposed in the studied articles are evaluated using a diverse array of performance benchmarks. Hence, a quantitative performance comparison may be infeasible. Even with the same benchmarks, it may not be possible to objectively compare the performance of techniques with different objectives, service requirements, isolation granularity, and other characteristics. Therefore, our criteria only provides a generalized perspective on the performance overhead incurred due to different design decisions. For an objective evaluation to be possible, future work in the development of standardized performance metrics for security isolation is necessary.

In this article, we attempt to provide four generalized categories based on the performance overhead: (1) *Negligible*, (2) *Practical*, (3) *Borderline*, and (4) *High*. We use these categories for comparison wherever appropriate (i.e., to compare identical approaches). Note that we are primarily concerned with what overhead will allow the security isolation technique to be incorporated into a *production* system. Many research prototypes do not meet this bar. Finally, some articles do not evaluate the performance overhead of their security isolation technique and, therefore, cannot be classified.

[1]In some cases, a system may use a combination of categories for different aspects of the security isolation.

Code requirement: Our second evaluation criterion considers the code requirement for incorporating the security isolation technique into a production system. For example, some security isolation techniques might use a compiler to extract information from source code to generate security isolation policy. Given recent advancements in reversing binaries, the binary of the subject to be isolated may be sufficient. However, some binary reversal techniques rely on symbols (i.e., relocation information, symbol table), which are sometimes stripped from the binary for performance or intellectual property reasons.

We break code requirement into four categories. First, *source code* indicates that the security isolation technique requires the source code for the subject or some part of the subject being isolated. This category does not explicitly indicate if the source code must be modified, but we touch on this characteristic in the evaluation where relevant. However, note that some systems (e.g., Berkeley Packet Filter [McCanne and Jacobson 1993] and some subsequent works such as Jitk [Wang et al. 2014]) may also be considered to fall into this category, although those techniques require code to be written in specific languages. Second, *binary with symbols* indicates both binary code and symbols are required for the security isolation technique to be deployed in practice. Similarly to the source code category, we do not distinguish binary code-rewriting. Third, *binary without symbols* indicates that some information from the binary is needed, but symbols are not required. This category is compatible with stripped binaries. Finally, *no code required* indicates that the security isolation technique does not require any *a priori* knowledge of the subject being isolated. This category is the most ideal case for deployment in production systems.

Security assurance: In contrast to the other evaluation criteria, it would be inadequate to evaluate security using a set of metrics. Instead, security is modeled using the properties of the "reference monitor" [Anderson 1972] that enforces the security isolation. The reference monitor must (1) provide *complete mediation*, (2) be *tamper-proof*, and (3) be *verifiable*. We now discuss the evaluation of the mechanism and policy aspects of security isolation techniques in terms of the properties of the reference monitor.

The enforcement mechanism provides complete mediation if it is uncircumventable with respect to the security isolation guarantees being provided, that is, provides complete mediation. The mechanism is tamper-proof if it can maintain its own integrity. These properties, especially resistance to tampering, are directly affected by the choice of the trusted computing base (TCB), that is, by the other components of the system that are trusted by the reference monitor.

To evaluate the security of the TCB, one may consider its size, both in terms of the number of entry points and the amount (and complexity) of its code. A larger TCB, in terms of the number of entry points (i.e., the Application programming interface (API) exposed) into privileged code, may have a larger attack surface. Further, the code size of the TCB (in terms of the lines of code), or the addition of code to existing API may also have direct influence on the scope for potential vulnerabilities [Rosenberg 2014]. Thus, a smaller TCB is generally desired for stronger security isolation guarantees.

Another approach would be to look at the actual contents of the TCB when comparing systems that provide security isolation at the same granularity (e.g., sub-application) and evaluate systems that trust the lower levels of the system architecture (e.g., hardware) as being more secure than those that trust the upper levels (e.g., the OS). For instance, Flicker [McCune et al. 2008] uses hardware to isolate the execution of critical sections of code within applications and is thus less prone to tampering than the Gazelle Web Browser [Wang et al. 2009] that relies on the OS to enforce similar sub-application isolation.

Table III. Comparisons Based on Evaluation Criteria with Regard to Enforcement Location

Categories (Subcategories)	Trusted Computing Base	Performance Overhead of common cases*	Code Requirement
Physical Host	Hardware	Negligible	No code modification required
Hardware component	Hardware, isolation technique framework	High	Application source code may be required
Supervisor			
Hypervisor	Hardware, isolation technique framework, or BIOS	Practical	Application source code or binary code may or may not require to be modified
Library OS	Hypervisor, OS, Library OS framework	High	Application code may require to be ported or recompiled
Container	OS, sandbox framework, Container Engine, Piece of Application Logic	Practical	No code modification required
Intra-application			
Code Rewriting	OS and Binary Writer	Practical	Modification to binary required.
Compiler	Operating system, compiler and runtime	Low	Source code modification may or may not be required
System Loading	OS kernel, isolation technique framework	High	Application code modification may or may not be required.

*Note that the performance overhead-based comparisons have been derived from the articles' descriptions, rather than an experimental study, and hence may be subjective and imprecise.

The primary means of evaluating the policy is verifiability. That is, in order to ensure correct isolation, the policy must be validated against the isolation goals defined for the system. Formal verification is both necessary and sufficient to ensure the soundness of an isolation policy and its consistency with the isolation goals. Unfortunately, formal verification for even slightly complex policies is hard. For instance, the simplest policy to formulate is one that ensures total security isolation, that is, denies all accesses in and out of the isolated environment. Policy generation as well as verification become more challenging when the policy rules are relaxed, and accesses are conditionally allowed. Further, approaches such as certifying compilers [Necula and Lee 1998] that incorporate the policy as a property of the language itself, and then formally verify the compiler output, can provide stronger security isolation guarantees than approaches with a complex and separate security isolation policy.

Finally, we note that formal analysis and verification of the TCB (e.g., seL4 [Klein et al. 2009]) may make it more trustworthy. At the same time, we note that the formal program verification of the reference monitor's implementation using tools such as Coq [Cornes et al. 1995] is an orthogonal problem and outside the scope of this work. The rest of the article uses the previously described criteria for analyzing the TCB of the discussed isolation techniques. We provide examples and describe the nuances of evaluating the TCB in detail in Section 3.3.

3. ISOLATION MECHANISM

This section discusses various security isolation mechanisms from the surveyed techniques. Different techniques enforce security isolation at different locations (e.g., supervisor, inside an application) and at varying granularities (e.g., guest OS, whole applications, some application components). Hence, we describe the isolation mechanism in two dimensions: (1) *enforcement location* and (2) *isolation granularity*.

In Tables III and IV, we present a general comparison of the isolation mechanism (i.e., the enforcement location and the isolation granularity respectively), based on

Table IV. Comparisons Based on Evaluation Criteria with Regard to Isolation Granularity

Categories	Trusted Computing Base	Performance Overhead of common cases*	Code Requirement
Guest OS	Hypervisor and Hardware	Practical	No application modification required, but guest OS modification may be required.
Application Group	OS, isolation technique framework	Borderline	No modification required.
Application	OS, hypervisor, isolation technique framework or service runtime	Practical	Many techniques need source code or binary modification
Sub-application	OS, isolation technique framework or hardware	Practical	Many techniques need source code or binary modification

*Note that the performance overhead-based comparisons have been derived from the articles' descriptions, rather than an experimental study, and hence may be subjective and imprecise.

evaluation criteria proposed in Section 2.3. The values in the tables are a purely generalized representation of each mechanism, based directly on our tradeoff discussions that follow each section. To make the table manageable with respect to the performance overhead criterion, we mainly account for the most common cases in each category. For example, among the four techniques that we study in the category of Library OS, Drawbridge [Porter et al. 2011], Mirage [Madhavapeddy et al. 2013], and Graphene [Tsai et al. 2014] incur a high performance overhead, while Bascule [Baumann et al. 2013] adds a negligible overhead. Therefore, we use the value "High" to stand for most of the cases, although there may exist some exceptions.

3.1. Enforcement Location

Security isolation is applied and enforced at different locations within the system. The enforcement location plays a critical role in determining and upholding security guarantees. Therefore, system designers must be aware of the benefits and tradeoffs of various enforcement locations.

As depicted in Figure 1, we identify four high-level categories that encompass the location of isolation enforcement: (1) *physical host*, (2) *hardware component*, (3) *supervisor*, and (4) *intra-application*. The supervisor category and intra-application category have sub-categories that will be described in detail in this subsection. These categories are loosely ordered from strongest to weakest guarantee of security isolation, which will be discussed at the end of this subsection.

Enforcement locations are not always mutually exclusive and one system can incorporate multiple locations to address specific threat models and system requirements. We now discuss each of these enforcement locations in detail citing specific examples.

Physical host: The strongest isolated systems are those that do no share any physical resources with others. Physical isolation is often used to prevent the threat of possible side-channel leaks [Zhang et al. 2011] and covert channels that can occur from sharing resources [Ristenpart et al. 2009]. Extremely sensitive materials may be physically isolated to prevent remote intrusion or unauthorized physical access to hardware components. File systems may also be physically isolated on different disks or over a network [Lu et al. 2014]. Networked systems have become a common requirement for modern communication, which could raise challenges for ensuring strong security isolation of physically separated systems. Still, networks can be monitored to uphold security isolation between physically isolated systems.

Hardware component: To support sharing of hardware resources among different users and applications, specialized hardware components have been developed to provide security isolation for shared resources. Those specialized hardware components can be either *passive* or *active*. Passive components can act as secure storage and provide tamper-proof logging and be used as building blocks in the active methods. For example, the Trusted Platform Module (TPM) [Trusted Computing Group 2011] provides tamper-resistant storage of keys that can be used to bootstrap trust in code that is loaded, in what is often a small trusted environment that operates separately from the main applications and OS.

Active components control critical system operations and exist as a property of execution. For example, Flicker [McCune et al. 2007, 2008] is an architecture that isolates sensitive code execution using hardware support from AMD' Secure Virtual Machine (SVM) architecture [AMD64 2005]. Intel's Trusted eXecution Technology (TXT) [Intel 2007] is similar in nature and function; that is, both SVM and TXT support Late Launch of a Virtual Machine Monitor (VMM) or Security Kernel to create a dynamic root of trust. Particularly, SVM launches the VMM by invoking the SKINIT instruction, which takes the physical memory address of Secure Loader Block as the only argument. Using this instruction, Flicker starts a special session to execute a marked critical section of code, that is, PAL, by suspending the underlying untrusted OS and all other software on the device. The OS is resumed at the end of the Flicker session. TrustVisor [McCune et al. 2010] is a special-purpose hypervisor that leverages similar techniques as Flicker but with a lower performance overhead. TrustVisor incurs low overhead because it employs a two-level integrity measurement approach, in which approach the measurements of TrustVisor are stored by physical TPM and TrustVisor in turns measures each PAL. This approach reduces the performance overhead that caused by frequent hardware support for Dynamic Root for Trusted Measurement (DRTM) in Flicker.

Research prototypes for protecting application code and data have motivated native support from processors. For example, Intel's Software Guard eXtentions (SGX) [McKeen et al. 2013] allows applications to protect their private code and data from privileged software (e.g., an OS daemon running as root). SGX generates protected software containers (i.e., enclaves) to prevent untrusted system software from accessing an application's sensitive code and data. An enclave is a protected area located inside application's address space, and the protected code of the application is loaded into an enclave after it is measured using hardware-based attestation. For confidentiality, enclave data are automatically encrypted when leaving the CPU package into platform memory. While SGX provides the necessary hardware support for protecting critical code and data, developers may find it difficult to identify sensitive data and all the operations that may work on that data. This is especially true in terms of complex applications. Additionally, correctly defining the enclave interface (i.e., the attack surface) may also be an overwhelming and error prone task for complex applications. Some isolation techniques such as Haven [Baumann et al. 2014] also leverage hardware support of Intel SGX for shielded execution of unmodified applications.

Finally, hardware security isolation guarantees of active components can not only allow applications to protect their private (and trusted) data and code but also can be extended to verify data on untrusted components. For example, in Cerium [Chen and Morris 2003], a tamper-resistant CPU and microkernel work cooperatively to resist hardware attacks. The microkernel partitions user-level processes' address spaces, while Cerium CPU makes use of memory protection to prevent from application accessing data located in other address spaces. Some data of the microkernel is stored in the secure CPU's cache to prevent tampering. The CPU identifies the microkernel by its kernel signature, which is the hash of microkernel's data. An identified microkernel can authenticate data and code in the untrusted Dynamic random-access memory (DRAM).

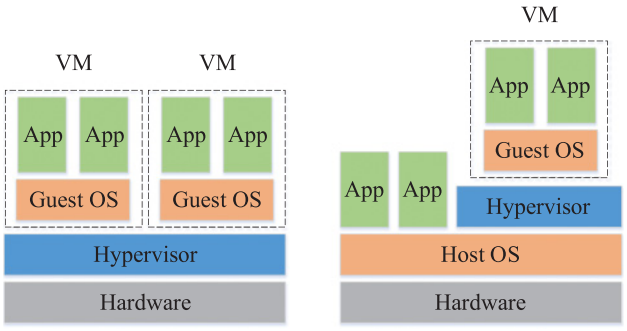
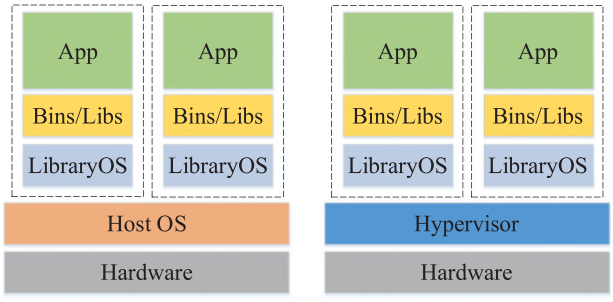


Fig. 3. Hypervisor.

Supervisor: A supervisor generally represents security isolation via a centralized entity that exists outside the isolated program. Typically, the isolated program is viewed as untrusted from the supervisor's perspective and therefore requires monitoring. Often, a supervisor is placed on the critical path of execution to intercept monitored executions and enforce security isolation policies. A supervisor may encompass enforcement that occurs in a kernel, hypervisor, or a virtualized environment like the Java Virtual Machine. There are mainly three groups of techniques to enable supervisor enforcement: (1) *hypervisor*, (2) *library OS*, and (3) *container*.

Hypervisors or virtual machine monitors (VMMs) can be software or hardware that create and host multiple virtual machines (VMs). They supervise guest VMs and trap access to critical resources such as protected instructions, memory, or CPU cycles [Chen et al. 2007; Cheng et al. 2013; Barham et al. 2003; Zhao et al. 2005]. There are two forms of hypervisors, namely type-1 and type-2, as shown in Figure 3. A type-1 hypervisor (bare-metal) runs directly on the host's hardware. A type-2 hypervisor (hosted) resides above a conventional host operating system and provides a full set of virtual hardware resources to the above guest OS. Further, a type-2 hypervisor translates all communications between the guest OS and the hardware. This translation may not be direct; that is, in some cases accesses to virtual devices are trapped to a host OS daemon that uses a device model to interpret the operation. The host OS daemon may then use host-level system calls to implement updates to the model. While a type-1 hypervisor's TCB may only contain the hypervisor code itself, the type-2 hypervisor also includes the underlying commodity OS in its TCB and hence provides weaker security guarantees.

The *Library OS* approach implements the OS functionality on which an application depends as a user-mode library [Engler et al. 1995; Ammons et al. 2007; Porter et al. 2011; Baumann et al. 2013; Madhavapeddy et al. 2013; Tsai et al. 2014]. The traditional OS kernel is refactored into a library, which implements only the OS system calls needed by the application as library function calls [Porter et al. 2011]. That is, a library OS usually executes with an interface restricted to a small set of host kernel Application Binary Interfaces. Without a shared system call interface, mutually untrusted applications running on library OS have little opportunity to interfere with each other. Therefore, by using library OS, the goal of securely isolating mutually untrusted, multi-process applications can be realized on the same host system, along with better guarantees of system and application integrity. Figure 4 shows two generic library OS architectures, which run on an unmodified host OSes or a hypervisor respectively. Further, compared to the approach of isolating applications in separate VMs, a library OS is lightweight and incurs far less performance overhead in terms of



(a) LibraryOS on Host OS (b) LibraryOS on Hypervisor

Fig. 4. LibraryOS.

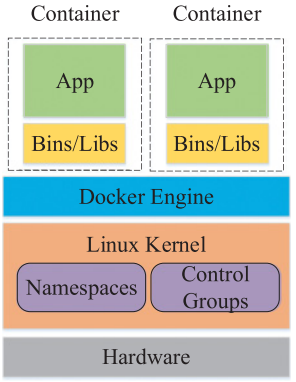


Fig. 5. Architectural overview of the Docker container.

the CPU, memory, and storage. At the same time, library OSes maintain backwards compatibility with applications [Porter et al. 2011] and support both single-process as well as multi-process applications [Tsai et al. 2014]. Finally, library OSes facilitate the secure and efficient isolation of third-party OS extensions [Baumann et al. 2013].

A *container* is a system abstraction developed to provide an alternate, restricted execution environment to an application or a group of applications. For instance, Linux Containers (LXC) [Helsley 2009] is an operating system level container-based virtualization technology that enables running multiple isolated virtualized environments on a Linux host. LXC uses Linux namespaces for separating shared resources among multiple containers as well as the host and provides applications the illusion of running on separate machines. Currently, Linux implements six different types of namespaces, namely mount namespace, UTS namespace, Inter-process communication (IPC) namespace, PID namespace, network namespace and user namespace. Global system resources are wrapped into abstractions through namespaces and applications that within the namespaces can only see the global resources associated with that namespace. Therefore, applications are able to have a unique view on the resources. Docker [Merkel 2014] is a container-based technique that incorporates Linux namespaces and control groups (cgroups) [Menage 2004] to create independent virtual environments. Docker containers are similar to LXC containers, and they have similar security features. Specifically, *Docker* uses the PID namespace to provide a contained application with its own virtual environment, where it can be the root process (i.e., PID 0). At the same time, in terms of the global PID namespace of the host OS, the container actually runs as a different PID and does not have systemwide root privilege. Similarly, Docker also provides the container with its own custom network namespace and file system. An application and its dependencies can be packaged into a Docker container, enabling flexibility and portability. Figure 5 provides a general overview of *Docker's* components within the OS. Another example is FreeBSD Jails [Kamp and Watson 2000], which partition the system into several independent controlled environments to isolate untrusted processes.

Many security mechanisms may use sandboxing (e.g., intercept all system calls from a sandboxed application), but may not necessarily be called containers. That is, containers are an application of sandboxing, the wider technique of confining a security principal (e.g., an application, a user) to a certain subset of the privileges available in the operating environment. For instance, TxBox [Jana et al. 2011] inspects system calls made by the program and its access to system resources and terminates program

execution on detecting a policy violation. While TxBox uses sandboxing, it does not really provide a virtual, customized, container-like environment to the application and hence cannot be called a container. Additionally, sandboxing may also be used in confining untrusted JavaScript code [Stefan et al. 2014; Giffin et al. 2012; Politz et al. 2011; Miller et al. 2008], although such systems may not be called containers. For example, Confinement with Origin Web Labels (COWL) [Stefan et al. 2014] sandboxes untrusted JavaScript in Web browsers to prevent it from leaking sensitive data.

Some real-world examples of sandboxes include secure computing mode [Corbet 2009], a sandboxing mechanism in a Linux kernel that restricts processes with limited number of system calls, and chroot in Linux systems, which restricts a program to a restricted root file-system (i.e., chroot jail), so the program is restricted from accessing files outside the environment. Android's Linux user identifier (UID)-based sandbox prevents applications from accessing protected resources not accessible to their users/groups and allows the OS to provide a clear separation between the private directories of applications. The Windows AppContainer [MSDN 2012] provides similar application sandboxing for Metro apps. Other examples of real-world sandboxes may include Qubes [Rutkowska 2012] from the Invisible Things Lab and Seatbelt [Miller et al. 2012] from Apple (discussed further in Section 4.3).

Now that we have discussed the types of supervisors, we discuss general properties that apply to all supervisors or subsets of the class.

In general, supervisors can leverage hardware features to enforce security isolation. For example, the System Management Mode (SMM) provided by x86 processors can be used to isolate workloads and configure hardware to ensure isolation of CPU execution [Azab et al. 2011]. Note that while the suitability of SMM in its current state is limited, it serves as a prominent example of leveraging hardware features for security isolation. Hardware features can be used to extend supervisor models to protect applications against malicious operating systems [McCune et al. 2008, 2010; Chen et al. 2008; Li et al. 2014; Yang and Shin 2008]. Additionally, software alone, without hardware support, can be used for supervisor-based enforcement. Native Client (NaCl) [Yee et al. 2009] provides supervisor enforcement that exists outside of an isolated and sandboxed binary. The sandbox limits API calls through well-defined interfaces that enforce a supervisor-based security isolation policy.

Supervisors can also be leveraged to achieve recursive isolation. The isolation goal behind nested approach may often be motivated by the need to reduce privileges at each subsequent level. For example, the two-tiered sandbox architecture proposed by Phung and Desmet [2012] allows web applications (themselves contained in a coarse-grained sandbox) to define a fine-grained application-specific isolation policy (i.e., the second tier) for untrusted JavaScript. Similar motivations, as well as the ability to run hypervisor-level rootkits and honeypots in a contained environment, have also led to nested hypervisor-based approaches. For example, the Turtles Project [Ben-Yehuda et al. 2010] efficiently runs multiple nested hypervisors such as Kernel-based Virtual Machine (KVM) and VMware as guest hypervisors, with associated virtual machines such as Linux and Windows as nested VMs. Similarly, Fluke [Ford et al. 1996] runs multiple vertically stacked VMs for securely running untrusted applications, using nested processes in a manner similar to LXC (specifically the Pid namespace).

As previously discussed, physical separation is often used to eliminate side-channels, but supervisors may also be made resistant to side-channels. For example, Deterland [Wu and Ford 2015] provides a hypervisor-based timing-channel-resistant approach. The supervisor in Deterland eliminates internal timing channels, that is, channels that allow guest VMs to learn information using shared resources such as the L1 cache, using system-enforced deterministic execution. Further, it rate-limits information leakage via external channels (e.g., a network connection) by applying

mitigation to each VM's external input/output. Similarly, Stefan et al. [2012] provide a language-based dynamic information flow control approach that is resistant to timing and termination channels. The supervisor in this case ensures that threads that observe secret values run with raised labels, including other threads that may observe the timing and termination of such secret threads.

Finally, supervisors can often be integrated into existing systems, which reduces the cost of deployment. Many supervisors extend the system call mechanism to enforce security isolation policy [Kim and Zeldovich 2013; Acharya and Raje 2000; Watson et al. 2010]. Some supervisor-based systems also require applications to be modified. TrustVisor [McCune et al. 2010] requires sensitive application logic to be extracted to be protected. Capsicum [Watson et al. 2010] adds a capability system by extending UNIX API but requires the application to be modified to use the capabilities. MiniBox [Li et al. 2014] does not require application modification but isolates an entire sensitive application providing a two-way sandbox.

Intra-application: Security isolation enforcement that occurs within the context of the target runtime component is referred to as intra-application location enforcement. This category encompasses those techniques and approaches that will modify runtime execution to include or remove some operations, and those operations may check or prevent accesses to isolated components. The most well-known examples that exhibit intra-application security isolation are Software Fault Isolation [Wahbe et al. 1994] and Inline Reference Monitors [Erlingsson 2003], which notably add policy enforcement as part of application runtime. However, implementations among these examples can differ based on techniques. In many cases, intra-application security isolation is paired with other location enforcement approaches to provide strong security guarantees [Yee et al. 2009; Kiriansky et al. 2002]. There are three groups of techniques used to enable intra-application security isolation enforcement: (1) *code rewriting techniques*, (2) *compiler techniques*, and (3) *system loading techniques*. Each group provides different capabilities and benefits when addressing design challenges such as performance, compatibility, and security guarantees.

Code rewriting represents approaches that apply policy by making static changes that enforce security isolation to existing code components. In these approaches, application source files, intermediate representation, or binaries are analyzed and transformed to include access checks or remove accesses to protect resources. Transformed sources are then reassembled into versions of binary applications that include security isolation checks. Code rewriting includes *static binary instrumentation*, which disassembles or decompiles a target binary into a source or intermediate language that is then transformed. One approach to isolate via code rewriting is to wrap critical or sensitive functions thereby including a layer of indirection to enforce security isolation policy. Similarly, function calls can also be replaced in a straightforward manner to hook restricted resource access [Singaravelu et al. 2006]. Code rewriting can occur on low-level languages such as assembly [Wahbe et al. 1994; McCamant and Morrisett 2006]. Pittsfield [McCamant and Morrisett 2006] rewrites assembly files to align Complex Instruction Set Computer (CISC) instructions to enable software fault isolation. Some instances of *dynamic binary instrumentation* rewrite control flow instructions to enforce security isolation policy at runtime. For instance, Kiriansky et al. [2002] rewrite and check all branch instructions and ensure security isolation checks are not bypassed. Further, Kiriansky et al. [2002] also use the RIO interpreter to perform instrumentation and enforce explicit policy when rewriting to protect the RIO interpreter itself from modification. In this case, the dynamic binary instrumentation also co-exists as a supervisor as security isolation policy protects the rewriting component itself.

Compiler-based intra-application security isolation adds and removes access to resources as part of the compilation process. In these approaches, security isolation policy

is a property of generated code enforced by compilers. One well-known compiler-based security isolation approach is through *type systems*, which rely on the properties of type-safe programming languages to assist the enforcement of security isolation at runtime. For instance, the type-safe properties of Java can ensure that programs can only access appropriate memory locations and perform known control transfers. Type systems have also been used to enforce well-defined interfaces and marshal shared data to isolate processes in software [Hunt and Larus 2007]. Isolation policy can be enforced by monitoring interfaces and IPC to verify capabilities performed on objects [Von Eicken et al. 1999].

Certifying compilers, which make use of the properties of a type-safe programming language, goes beyond type systems by including a proof that generated code satisfies a set of security rules. Code can then be statically verified to ensure that a program follows a given security isolation policy. Satisfying a set of security rules can be an iterative process. Certification that does not satisfy a correctness claim may lead to application refinement and recompilation until claims are met [Holzmann 1997].

Finally, compiler-based security isolation can also be applied to non-type safe languages. For example, Native Client [Yee et al. 2009] provides a constrained execution environment for the C and C++ languages and sandboxes the runtime component by restricting memory access within the sandboxed binary. It uses static analysis to detect security defects in untrusted code and utilizes segmented memory to limit data and instruction memory reference with the help of hardware range checks. Native Client also limits instructions available at runtime and requires that all APIs flow through a well-defined interface to enable supervisor capabilities. For its enforcement, Native Client inserts instructions to mask off address ranges before any write instruction. This property is useful to protect a small trusted region of code within the address space, and trampolines can be placed in that region to ensure important API calls can have mediation and security checks. Thus, Native Client's enforcement is in fact present in three locations (i.e., hardware component, supervisor, and intra-application) to provide strong intra-application security isolation security guarantees.

System loading performs intra-application security isolation by forcing target applications to use customized libraries that perform access control checks. Applications can be forced to use security isolation libraries seamlessly without the knowledge of or modification to the main program logic. Customized libraries can be loaded as part of the library loading process or by repackaging security isolation libraries with the target application. Aurasium [Xu et al. 2012] repackages arbitrary Android applications by inserting instrumentation code into applications to attach sandboxing and policy enforcement code. Aurasium also leverages monitor code to intercept application's interaction with operating system so the behaviors of applications are monitored for security and privacy.

While system loading is generally insecure because a `syscall` instruction in the untrusted code could circumvent enforcement, separating the untrusted code into another sandboxed process may provide complete mediation. That is, the sandboxed application may be deployed in an unprivileged process, with all protected library calls being redirected to a different reference monitor process that performs the access control check. For example, consider Boxify [Backes et al. 2015], a security mechanism deployed as an Android application that can load and isolate untrusted third-party applications. Boxify uses Android's "isolated process" security feature to encapsulate untrusted Android application code in an unprivileged (i.e., isolated) process. Any syscalls or Android API calls made by the application code are forwarded to Boxify's reference monitor (i.e., the "Broker"). A permitted call is executed in the broker's context and the result is returned to the isolated application process. Note that Boxify is secure as it uses Android's isolated process abstraction, which is backed by Linux's UID enforcement as

well as Security-Enhanced Linux (SELinux), that is, each isolated process has a separate Linux UID and the isolated process type has a dedicated sandbox in SELinux.

Discussion: Ideally, security isolation enforcement should incur low performance overhead, be largely compatible, and provide strong security isolation guarantees. However, satisfying the three properties simultaneously is challenging. For instance, while supervisors as reference monitors offer complete mediation (for isolating the layers above) and hence strong security isolation guarantees, they may incur high performance overheads for commonly used functionality [Provos 2003; Kim and Zeldovich 2013; Chen et al. 2008; Acharya and Raje 2000; Hofmann et al. 2013; Jana et al. 2011; Porter et al. 2011; McCune et al. 2010; Yang and Shin 2008]. This is mainly due to the context switch, that is, from the enforced program's context to the reference monitor's (e.g., kernel's) context. In some cases, supervisors also adversely affect backwards compatibility by requiring applications to modify their source code [McCune et al. 2010; Watson et al. 2010; Hunt et al. 2007; Zhou et al. 2014b] or binaries [Hofmann et al. 2013; Yang and Shin 2008].

The engineering advantages as well as the enforcement context (i.e., policy complexity) at various enforcement locations often directly impact system performance, security, and compatibility. For instance, hardware components can often be fine tuned to provide security isolation enforcement with minimal added overhead. Hence, techniques that use hardware and especially specific hardware features meant for security isolation (e.g., Intel SGX) may generally exhibit low performance overheads. Further, techniques that only rely on the hardware generally have smaller TCBs and provide stronger reference monitor guarantees. Yet, hardware can only provide coarse-grained isolation, as it lacks the context from the higher layers (e.g., the kernel, application layer). If hardware support is to be used for fine-grained isolation (e.g., for isolating parts of an application), then the context (i.e., the critical section to be isolated) often has to be explicitly specified to guide the enforcement (e.g., Flicker [McCune et al. 2008]). This requirement in turn sacrifices on backwards compatibility.

On the contrary, supervisors residing in the OS can generally be more precise than the hardware for the goal of protecting OS resources and isolating entire applications, without requiring application instrumentation. Such approaches may still suffer from higher performance overheads (relative to hardware). Further, supervisor-based approaches reside in the OS or hypervisor and hence often require OS modification to exist or upgrade, which may be avoided by the use of intra-application reference monitors. For example, compiler-based intra-application security isolation that use type-safe languages can provide type safety without OS modification at the cost of additional runtime overhead for the type enforcement. Similarly, dynamic binary instrumentation, which enforces policy on runtime operation, also suffers from performance overhead well beyond supervisor-based approaches [Kiriansky et al. 2002].

As discussed in Section 2.3, the potential for vulnerabilities in privileged code increases with the size of the TCB (in terms of the code size as well as the attack surface, that is, the API "entry points"). For example, SICE uses a supervisor consisting of 300 Lines of code (LOC) to manage security isolation of components in System Management Mode [Azab et al. 2011]. TrustVisor [McCune et al. 2010] uses a thin hypervisor to isolate small PAL from untrusted operating systems. Such techniques may be said to be more secure than ones that use higher levels of the OS architecture for enforcement (i.e., have a larger attack surface). Further, microkernels isolate sensitive operations into small kernel components and allows kernel drivers to execute with fewer privileges, thereby having a smaller TCB than monolithic kernels. Generally, type-1 hypervisors are believed to be more secure than operating systems that can contain orders of magnitude more code [Barham et al. 2003]. However, type-2 hypervisors include the entire host operating system in the TCB making it difficult to argue the security guarantees.

Note that for supervisors that exist within the address space of the isolated component, such as a sandbox (e.g., Native Client), the TCB includes software running within the process address space outside of the sandbox.

Reconfiguring system architecture can incur compatibility challenges, which may be avoided at the cost of security. That is, as we described before, intra-application approaches do not require OS or architecture modification but may trade off security (and also performance overhead) in most cases. The primary reason for this is that the TCB for such approaches includes a portion of the untrusted application itself. Compiler-based security isolation is an exception, as the code generated has security isolation policy enforced as a property of the compiler and the language.

Finally, prior research has shown that combining multiple enforcement locations provides significant advantages in performance, security advantages, and backwards compatibility. Consider Native Client [Yee et al. 2009] as an example of a system that enhances precision while maintaining security through a combination of enforcement locations. The isolation goal behind Native Client is to sandbox untrusted application components. For this, Native Client deploys its reference monitor as a *supervisor*, but within the process address space of the application, that is, *intra-application*. Additionally, Native Client prevents a potentially drastic performance hit due to its intra-application supervisor access control checks through its use of x86 segmented memory. While Native Cleint requires application source code to be compiled with a trusted compiler, it does not require the OS to be modified and hence may provide a compatibility advantage in scenarios where the OS may not be easily replaced or modified. To summarize, Native Client [Yee et al. 2009] combines the security advantages of a supervisor-like enforcement, specifically tamper resistance, with the precision benefits of intra-application placement. SIESTA [Hicks et al. 2007] also provides similar advantages in terms of security and precision, but, in contrast with Native Client, it requires OS modification to enable SELinux.

3.2. Isolation Granularity

The isolation granularity determines the location of the protection domain and the objects isolated by the enforcement mechanism. In this section, we focus on the high-level, logical granularities of security isolation based on the types of isolated objects we encountered during our survey. The four levels of isolation object granularity are as follows: (1) *guest OS*, (2) *application group*, (3) *application*, and (4) *sub-application*.

While some security isolation techniques may be limited to be enforced at a specific granularity, some techniques may be enforceable at more than one granularity, with potential changes to their enforcement location and policy. For example, the SICE [Azab et al. 2011] framework provides hardware-level security isolation and protection, at the granularity of an abstract *workload*, with the objective of protecting sensitive workloads while also running untrusted workloads on the same hardware. The isolated workload is a user-defined system, which may be a simple program, a collection of programs, or a complete virtual machine running its own guest OS. Thus, as the object to be isolated is user defined, SICE can be said to operate at the operating system, multi-application, and application granularities.

We now describe each of the aforementioned granularities in detail.

Guest OS: At this granularity, multiple operating systems sharing a resource (e.g., hardware) are isolated from each other, often running as *guest* operating systems on the shared resource. Such security isolation is most commonly implemented using virtualization techniques. Some approaches, such as Xen [Barham et al. 2003], allow guest OSes to directly share hardware. The guest OSs may even be run on virtual machines directly on the host OS, as in User-mode Linux (UML) [Dike et al. 2001].

OpenVZ [Kolyshkin 2006] offers an interesting middle ground, whereby the guests share the microkernel but individually implement all other essential functionality (e.g., file system, network interfaces, and user interfaces).

Application group: At this granularity, groups of applications form the *subject* that is isolated from other parts of the operating system. Based on the security isolation goals, the enforcement may span different areas of the shared operating system from the kernel to the user interfaces. Approaches supporting this granularity often group applications into *containers* based on a high-level policy (e.g., separate enterprise applications from user applications, group by similar functionality, or hierarchical groupings), which are then isolated using policy controlled sandboxes (e.g., MAPBox [Acharya and Raje 2000], Linux Container [Helsley 2009]). For instance, MAPBox [Acharya and Raje 2000] classifies applications into functionality-based classes (e.g., reader, compiler), based on the information (i.e., labels) provided by the application provider. The user then assigns a sandbox policy for each corresponding label. Other approaches also use virtualization to build and enforce containers (e.g., Terra [Garfinkel et al. 2003], PeaPod [Potter et al. 2007], FreeBSD Jail [Kamp and Watson 2000]), where one or more applications are started in a virtual environment isolated from the rest of the system. For instance, using FreeBSD Jail, a BSD administrator can partition the machine into several independent jails with their own superuser accounts, as well as protect the overall environment from the jailed superusers. This is achieved through several limitations on the processes running in the jails, such as access to a subset of the file system, specific IP addresses, and the ability to make privileged system calls.

It is possible to enforce application granularity security isolation using the techniques developed for application group granularity by simply limiting the containers to contain exactly one application. On the other hand, extending an application granularity technique to a multi-application level may require more effort, specifically in defining and isolating the container abstraction.

Application: At this granularity, individual applications are isolated from the rest of the system, that is, other applications, as well as parts of the OS.

Application-level security isolation approaches can be divided along the lines of the objects they protect. Some approaches protect the application itself from an untrusted OS and other untrusted applications (e.g., InkTag [Hofmann et al. 2013], Lacuna [Dunn et al. 2012], PrivExec [Onarlioglu et al. 2013], Cerium [Chen and Morris 2003], Over-Shadow [Chen et al. 2008], BROKULOS [Santos et al. 2012], and Ironclad apps [Hawblitzel et al. 2014]). For instance, InkTag [Hofmann et al. 2013] provides security guarantees via a hypervisor to protect trusted applications from the untrusted underlying operating system. Trusted application code runs in a high-assurance process (HAP), and the process context (registers) and address space of HAP are isolated from the operating system. InkTag also enables sharing data between trusted secure applications without interference from operating system. Similarly, PrivExec [Onarlioglu et al. 2013] provides a private execution mode to protect the secrecy of application data on storage. Executing an application process in the private mode ensures that the application's data written to storage would not be recoverable after execution. This guarantee is enforced by encrypting the data written to storage with an ephemeral private execution key that is bound to the group of processes executing in private mode; the key is removed from memory after execution. Some approaches, such as BROKULOS [Santos et al. 2012], protect applications from an untrusted system administrator and software running with elevated privileges by logically separating the management and information security mechanisms in the OS. BROKULOS uses a set of trusted programs, known as brokers, to provide the administrator with management access, without superuser privileges that could compromise application data. On

the contrary, approaches such as Ironclad apps [Hawblitzel et al. 2014] use low-level software verification to ensure that all the software on a server running an Ironclad app behaves in a trusted manner.

Additionally, some approaches isolate untrusted and potentially malicious applications from the rest of the system, with the goal of protecting the OS, other applications, and, eventually, the user (e.g., TxBox [Jana et al. 2011], MBOX [Kim and Zeldovich 2013], and Polaris [Stiegler et al. 2006]). For example, TxBox [Jana et al. 2011] confines untrusted programs via system transactions with the help of a kernel-level security monitor and a user-level policy manager. Before a transaction of the sandboxed program commits, the security monitor inspects the policy and aborts the transaction if a violation is detected, rolling back the system to a state before the program's execution.

Finally, approaches such as Minibox [Li et al. 2014] can provide two-way protection, that is, protect mutually distrustful applications and the kernel from each other. Minibox uses a combination of hypervisor-based memory isolation (i.e., TrustVisor) and a sandbox (i.e., Native Client). In Minibox, TrustVisor protects security-sensitive applications by creating separate virtual memory regions for the OS and applications, while Native Client protects the OS from non-sensitive applications by sandboxing them.

Sub-application: At this granularity, different logical components of an application may be isolated from the rest of the application and the OS. Sub-application security isolation approaches secure processes or threads with access to sensitive data for both integrity and secrecy guarantees (e.g., Capsicum [Watson et al. 2010] and PREC [Ho et al. 2014]). For example, in PREC [Ho et al. 2014], system calls from untrusted components are identified and executed in isolated threads. Others, such as Flicker [McCune et al. 2008] and TrustVisor [McCune et al. 2010], protect critical sections of code. In Flicker, security-sensitive application code is executed in an isolated execution context, while all the other software (e.g., the OS, other application code) is suspended. Similarly, TrustVisor [McCune et al. 2010] executes selected selected code blocks in an execution environment isolated from the operating system and other untrusted application code to guarantee data secrecy and integrity, as well as execution integrity. Another example of sub-application-level isolation is *application domains*, which provide isolation for Microsoft's .NET Common Language Runtime [Clayton 2013]. Application domains are usually created by default by the runtime host (e.g., ASP.NET, Microsoft Internet Explorer), which bootstraps the Common Language Runtime into a process and executes user code within an application domain.

Sub-application security isolation may also be used to sandbox parts of the application that are untrusted in order to protect the rest of the application and the operating system [Kiriansky et al. 2002; Wahbe et al. 1994; Pearce et al. 2012; Shekhar et al. 2012; Ford and Cox 2008]. For example, program shepherding [Kiriansky et al. 2002] monitors control flow transfers to enforce security policy. That is, malicious code is prevented from executing, and libraries are entered only through declared entry points. Shepherding also leverages sandbox to check program operations with customized security policies. AdDroid [Pearce et al. 2012] and AdSplit [Shekhar et al. 2012] are examples that attempt to isolate untrusted third-party ad libraries operating in the application's context, thus preventing the libraries from executing with the application's authority. Some approaches also target web browsers, wherein web security principals operate in isolated security contexts [Wang et al. 2009]. Further, approaches such as Self-spreading Transport Protocols (STP) [Patel et al. 2003] also sandbox application code that must execute in the OS kernel (e.g., application-specific TCP implementation). Similarly, SPIN [Bershad et al. 1995] provides an extensible microkernel that allows applications to run application-specific services in the kernel while sandboxing such services from other applications and the trusted portions of the kernel.

Finally, some approaches span entire data or untrusted code. For example, Virtual eXecutable Archive (VXA) is an architecture for data compression. It runs a decoder's code in a specialized virtual machine, making the decoder run safely even in the presence of malicious code inside an archive. Similar examples include Wimpy Kernels [Zhou et al. 2014b] and Embassies [Howell et al. 2013].

Discussion: The granularity of security isolation is directly proportional to the flexibility provided by the isolation technique, as well as the magnitude of policy-management needed from the concerned stakeholders. Fine-grained security isolation (e.g., sub-application) provides more flexibility in terms of enforcing security isolation policy but often requires multiple stakeholders (e.g., the user, system administrator, developer) to specify policies. On the contrary, coarse-grained security isolation (e.g., application level) often has well-defined policy but limited knowledge of higher granularity semantics and, hence, limited flexibility. For example, in order to protect against potentially malicious ad libraries bundled with applications, an application-level policy defined by the OS may isolate and restrict the entire application, whereas a sub-application-level policy may recognize and isolate only the library, allowing the user to make full use of the application otherwise. Additionally, the sub-application approach may require some cooperation from the application developer, such as the use of a manifest file or other artifacts that clearly specify the location of the library code.

If the isolation technique benefits the security of the application, as in the case of Capsicum [Watson et al. 2010], then it may be acceptable to expect some form of policy specification from the developer. On the other hand, if the security isolation does not benefit the application (e.g., the developer is assumed adversarial), then developer participation may not be appropriate. Furthermore, requiring recompilation or refactoring of the application can affect the backward compatibility of the platform. To summarize, the coarser-grained security isolation approaches are feasible but often lack the flexibility of the finer-grained approaches. On the other hand, the finer-grained approaches face challenges such as of backward compatibility and policy management.

Additionally, while the performance overhead may be connected to the granularity of isolation, it is largely influenced by how the isolation is achieved. For instance, sub-application isolation may be achieved through an unoptimized inline reference monitor approach (high overhead) or a hardware supported approach (e.g., Flicker, low overhead). Hence, we often observe disparity in the overheads among approaches that fall in the same isolation granularity. For example, some security isolation techniques in the application and sub-application levels incur practical to high overhead [Hofmann et al. 2013; Onarlioglu et al. 2013; Chen et al. 2008; Kiriansky et al. 2002; Ho et al. 2014; McCamant and Morrisett 2006], while some incur negligible or borderline overhead [Watson et al. 2010; Howell et al. 2013]. As we described before, overheads within approaches belonging to a particular isolation granularity may differ based on the approach used to achieve the isolation, specifically the location of the reference monitor.

Finally, for code modification requirements, the guest OS granularity usually does not require code changes [Barham et al. 2003; Dike et al. 2001]. One exception is SICE [Azab et al. 2011], which requires the System Management Interrupt (SMI) handler to be modified to include SICE's code. The hardware management functions provided by the legacy host must be modified to use this interface to provide services to the isolated environment. There are also some cases in application group granularity that need no code modification [Potter et al. 2007; Acharya and Raje 2000; Kamp and Watson 2000; Garfinkel et al. 2003; Ammons et al. 2007]. However, in the other isolation granularities, many techniques require source code or binary code modifications and a few require no modification.

3.3. Caveats of TCB Evaluation

Section 2.3 provides a general description on the implications of TCB size on the attack surface and eventually the integrity of the reference monitor. In this section, we discuss the two nuances that must be considered when analyzing the TCB of a system.

1. For a correct evaluation of TCB size, both the shared as well as the private TCB components must be considered. The TCB may be divided into two logical sub-areas, the "shared" and "private" TCB. Parts of the system that are trusted, but shared among isolated entities, constitute the shared portion of the TCB (e.g., the OS kernel). Parts of an isolation system that are trusted by individual isolated entities form the private TCB (e.g., the application's code). For instance, Library OSes reduce the size of the shared TCB at the cost of increasing the size of the private TCB. At the same time, the security of a capability system (e.g., Capsicum [Watson et al. 2010]) depends on the code of the application using capabilities (e.g., the private TCB), as well as the OS kernel that enforces the capabilities (e.g., the shared TCB). Thus, the TCB size increases with the size of the entity to be protected. If that entity is a complex application with a large number of components, then its overall TCB size will naturally be larger.

To summarize, we observe that in some systems the private TCB size increases in favor of decreasing the shared TCB size, or vice versa, while in other systems (e.g., Capsicum), the two constituent TCBs may grow independently of one another. Regardless of the relation between the two constituents, system designers must take care to consider both the shared as well as the private TCB for a correct evaluation of the attack surface.

2. The composition of the TCB is only meaningful with respect to the eventual isolation goals of the system, which in turn influence the threat model. The isolation goals dictate the subjects and objects that can be trusted and the ones that cannot. The ones that can be trusted should form the TCB. For instance, in Xaor [Colp et al. 2011], the hypervisor as well as the Control VM (domain 0) are trusted and, hence, should form the complete TCB.

Further, a microkernel architecture may not necessarily have a smaller TCB for the isolation goal of separating applications if user-level services are shared between applications. On the contrary, if the goal is to protect kernel services from applications and the vulnerable user-level daemons, then the microkernel approach may be said to have a smaller TCB, as only the kernel will be trusted.

Similarly, since Flicker's goal is to isolate a small critical section of code from the rest of the software on the system, the said software (i.e., the OS, other applications, other components of the same application) may not be trusted and, hence, cannot be included in the TCB. Hence, Flicker makes the TCB smaller for the goal of isolating a critical section of an application; in comparison to a library OS or VM-based approach. At the same time, since Flicker does not claim to provide application-granularity isolation, making the TCB of the entire application smaller is irrelevant from its perspective. Therefore, to accurately describe the composition of the TCB, system designers must consider the isolation goals and objectives of the desired system.

4. ISOLATION POLICY

A security policy describes the actions or executions that may or may not be permitted [Schneider 2000]. For instance, a security policy used for access control may define the ability of a subject (i.e., the security principal) to access objects in the system. Similarly, an information flow control policy may place restrictions on the data flow between subjects and objects in the system. A policy used for security isolation defines the separation between the isolated subjects and objects in the system.

For example, the security isolation policy may determine the precision of the restrictions placed on isolated entities. An imprecise (i.e., coarse-grained) policy may impose complete isolation, that is, may prevent the isolated entities from all external communication. Isolation policies that demand physical separation (e.g., Lu et al. [2014]) may commonly follow this policy. From the policy perspective, policies that allow precision in terms of the resource accesses are more interesting but also complex. For example, the SELinux policy may regulate the resources accessible to a root daemon based on the policy administrator's understanding of the least privilege it requires. The policy provides precision as the daemon may be completely isolated (i.e., no resource access) or may have access to a subset of all resources or to all the resources. Additionally, some policies may not just regulate resource access but also provide provisions for enforcing how the resource is used and are even more precise. For instance, information flow control policies protect the secrecy and integrity of data beyond the point of access; that is, they control the flow (and hence sharing) of data. Unfortunately, the complexity of policies increases with precision.

Further, throughout our survey, we encountered numerous examples of how policies are actually expressed, that is, the isolation policy may be implicit and programed into the mechanism itself, contained in a rule-set describing the constraints on access to data or code, or defined in terms of an hierarchy of security labels, along with the assignment of labels to subjects and objects (e.g., for information flow control). This section discusses various such policy examples, along the three dimensions of (1) *policy generation*, (2) *policy configurability*, and (3) *policy lifetime*.

4.1. Policy Generation

There are two broad approaches for policy generation: automatic and manual. While automatic methods may use static or dynamic analysis to determine the constraints to place on the subject of isolation, manual methods require human involvement.

Automatic: Security isolation policies can be automatically generated using *dynamic analysis* or *static analysis* of the target application's code. Automatic generation is often used when enforcing a *least privilege* policy, because manually specifying each access can be tedious. Representative examples for each approach follow.

Dynamic analysis can be used to learn an application's runtime behavior during a training session. Systrace [Provos 2003] is an example approach that logs system calls during a training period. The system calls form a white list that defines the program's policy. However, there is a risk of the policy being too restrictive. If there are sections of code that were not covered during training or trained in wrong environment, then they may cause false alarms during normal use. This can be addressed by allowing a human to make policy changes when false alarms are detected. One common problem with its dynamic analysis is that the policy generation includes malicious functionality. Therefore special care must be taken to ensure that policy generation is performed in a sanitized environment. Such techniques may often involve malware detection during the training phase to weed out applications that exhibit malicious behavior to prevent known malicious behavior from being included in the white list.

Static analysis can automatically generate a policy by searching through an application's code for system calls. System calls detected in the application are added to a white-list policy. Static analysis may have high false positives (e.g., the white list may also account for dead code, that is, code that may never run). Additionally, static analysis may lack context for system call use, resulting in overly permissive policy. For example, runtime conditions may influence the argument to a system call. These cases may require the automated policy generator make assumptions that reduce its accuracy. Human experts can augment this technique by removing dead

code, reducing ambiguity, and removing any privileges that do not seem justified. Rajagopalan et al. [2006] implement this approach to generate policies for system call monitoring.

Manual: Manual policy generation techniques can be discussed in terms of which decisions must be made manually. Manual generation can be done at various levels of complexity, which allows a tradeoff of ease of use and security guarantees. It is also important to consider which stakeholder must make these decisions.

System calls are a common feature listed in security isolation policies. Selecting which system calls to regulate and how to regulate them can significantly improve an application's security. Some system calls are more likely to be abused than others. They can also be regulated more effectively if different policies are specified for handling different types of system calls. MiniBox [Li et al. 2014] uses this approach as part of a two-way sandbox that protects applications and the operating system from each other.

Sections of program may be easier to protect than isolating an entire program. Such a section is referred to as a PAL. When the PAL is executed, more strict security isolation is enforced than when the rest of the program runs. Isolating only the PAL can significantly reduce the amount of code that must be protected and trusted. However, it is up to users or developers to determine which sections to protect and how to protect them. TrustVisor [McCune et al. 2010] use this approach to reduce the size of their trusted code base and reduce their security isolation system's overhead. Capsicum [Watson et al. 2010] allows application processes to use a system call that limits the privilege of the process. The rest of the application's processes do not need the same level of protection and can run normally.

Coarse application-level policies can reduce the complexity of manually generating a policy. A human must still determine which policy to use for each application, but the decision is simplified. Due to their coarseness, these policies may be too restrictive or overly permissive. One example is Android's permission model, which is described in detail in Enck et al. [2009]. Developers choose options such as camera, microphone, or contacts instead of listing system calls and file accesses. End-users are also more likely to understand these high-level permissions when they review the application before installation. Further, Cerium [Chen and Morris 2003] allows three security levels for an application, that is, (a) no protection, (b) authentication only, or (c) copy-protection and authentication. Authentication guarantees that the application was not manipulated or forged. Copy protection prevents the application's code and data from being accessed by malware. Finally, MAPbox [Acharya and Raje 2000] provides labels (e.g., compiler, editor, browser) that represent generic application types. Once an application is assigned a label, it can be placed in a sandbox suitable for that application type.

Discussion: The techniques discussed above present several tradeoffs in terms of performance, ease of use, and security.

Policy generation for achieving certain generally applying security principles (e.g., least privilege) may be harder to automate than for properties whose semantics may be defined specific to the program or situation (e.g., intra-program information flow). That is, policies based on low-level rules such as system call and file access often seek to approximate *least privilege*. Unfortunately, ground truth for least privilege is hard to define, and therefore correctness cannot be verified automatically. As least privilege is subjective, these policies are dependent on the expertise of the human generating the policy. On the contrary, in systems where security principals define information flow constraints their own data, reasoning about the allowable flows may be more tractable. For example, SWIM [Harris et al. 2010] instruments program source code with the system policy generated from a high-level policy and the program supplied

by the programmer. The input policy specifies both allowed and denied flows within the program, which are feasible for the developer to define as the flows are particular to the developer's application (unlike least privilege, which is a generally applying principle). For each process in the program, the programmer specifies a "template" (i.e., group), and the input policy specifies the flows between templates. SWIM transforms the input program and policy into logical constraints; a solution to such constraints would correspond to an instrumented version of the program that satisfies the policy. SWIM is formally correct, that is, it only produces an instrumented program if formal instrumentation conditions that show if one program is a correct instrumentation of another are satisfied.

Automated methods are easier to use than manual methods. However, limitations of static and dynamic analysis impact their accuracy. Dynamic analysis can produce policies that deny required functionality (i.e., false positives). Static analysis produces policy that has unnecessary privileges due to dead code or insufficient runtime context during analysis. These limitations can be mitigated if a skilled administrator checks the policy and corrects mistakes. However, this process may be time-consuming and likely to suffer from human error. Further, policies may need updating after significant changes in the system or application.

The developer's cooperation, for example, marking sensitive modules/data during development, can be effective to determine the security of applications, but such cooperation is often challenging and impractical. For example, isolating a subsection of a program can improve performance and security, but developers must identify the sensitive PALs. Therefore, this approach is difficult to apply to a large number of applications. Widespread adoption and backwards compatibility with legacy applications are unlikely when developer cooperation is required.

Coarse application-level policies may be easier to enforce, but their coarseness reduces the security guarantees they can provide. If programs that are too similar use almost the same list of system calls, then they could be isolated with the same coarse system-call-based policy. However, the policy will inevitably be either too permissive or too restrictive for at least one of the programs due to their different needs.

The performance overhead of policies is influenced by the policy generation approach. Dynamic analysis may incur a high overhead since applications are executed and their behaviors are recorded in order to generate policies for them. When a new system call is encountered, and it is not specified in the policy, a new policy statement will be added to match this system call [Provos 2003]. Manually generating policies based on system call may also bring high overhead [Li et al. 2014]. However, manually specifying a section of programs can reduce overhead to borderline [McCune et al. 2010] or even the negligible [Watson et al. 2010].

Finally, using dynamic analysis to automatically generate policy commonly requires no code modification (e.g., Systrace [Provos 2003]). In contrast, manual policy generation using sections of programs [McCune et al. 2010; Watson et al. 2010] requires source code modification.

4.2. Policy Configurability

Policy configurability captures the ability for the isolation policy to change after the system is deployed. The ability of the policy to change may allow isolation to dynamically adapt to evolving attack scenarios.

We divide policies into two categories based on their configurability: (1) *reconfigurable* and (2) *non-reconfigurable*. We further divide the reconfigurable category into automatic and manual subcategories. Note that we consider the aspect of configurability after the initial isolation is enabled; whereas Section 4.3 discusses the separate dimension of the policy lifetime.

Reconfigurable: An isolation policy is reconfigurable if it can change after the system is deployed. Policies that are defined via a configuration file are inherently reconfigurable. However, a configuration file is not a prerequisite for reconfigurability. It is useful to describe reconfigurability with respect to how the configuration change occurs: automatic or manual.

Automatic reconfiguration changes the isolation policy during runtime based on some pre-defined condition (e.g., event trigger). Prior research on security isolation policy has limited cases of automatic reconfigurability. One example is Capsicum [Watson et al. 2010], which extends the file descriptor abstraction by adding capability support to a UNIX system. Further, unlike traditional access control using Access Control Lists, capability systems such as EROS [Shapiro et al. 1999] and its predecessor KeyKOS [Frantz 1988] make isolation dynamics based on the runtime conditions. For instance, Capsicum supports dynamic delegation of capabilities to subjects, which changes the privileges (and hence the policy) of the subject automatically at runtime.

Information flow control (IFC) [Denning 1976] systems provide another example. The goal of IFC is to preserve a certain desired flow of classified data within the system. Thus, only communication that satisfies the IFC guarantees required by the policy is allowed. Some IFC systems allow all subjects to read confidential data but also propagate the restrictions on the data to the subject (similar to taint tracking). Hence, a subject's current isolation policy is automatically configured on the previous files and processes with which it interacted and is reconfigured every time it reads data. More recent decentralized information flow control (DIFC) approaches such as Asbestos [Vandebogart et al. 2007], HiStar [Zeldovich et al. 2006], Flume [Krohn et al. 2007], and Fabric [Liu et al. 2009] allow security principals to create and apply labels to their own data at runtime. Such security principals may delegate the ability to read, write, or export their data to subjects in the system, automatically configuring the isolation policies of the subjects (similar to capabilities).

Unlike systems that prefer explicitly defined flows (e.g., Flume, HiStar), Asbestos uses floating labels (i.e., implicit label propagation) to allow seamless communication. That is, communication between two subjects results in the caller's label and its associated restrictions propagating to the callee. Such implicit label propagation facilitates more dynamic communication primitives (e.g., user-directed ad-hoc communication) but may also lead to data leaks [Denning 1976; Krohn and Tromer 2009] in DIFC operating systems that rely purely on dynamic enforcement (e.g., Asbestos). Finally, the LoNet architecture [Johansen et al. 2015] describes the use of meta-code attached to data to enforce various kinds of security or data-use policies. Meta-code is attached to files and can not only control use of the data but also perform other functions such as logging. Additionally, in a manner similar to implicit label propagation in IFC systems such as Asbestos, meta-code is implicitly propagated to derived data by default. This default may be overridden if needed; that is, a file's meta-code may specify new meta-code for any derived data to inherit.

Manual reconfiguration is the most common type of policy reconfigurability. Frequently, manual policy reconfiguration requires restarting a process or, as in the case of SELinux [National Security Agency 2009], restarting the operating system.

For some types of manual policy reconfiguration, the policy is specified by the application user. For example, Cerium [Chen and Morris 2003] is a trusted computing architecture that protects a program's instructions and data with a tamper-resistant CPU. However, to isolate a program execution, the program needs to specify its protection policy in its header files so Cerium can know which level of protection should be applied to its data and instructions. Another example is Singularity, which provides Software-Isolated Processes (SIP) for Manifest-Based Programs (MBP). A manifest

describes the resources necessary for an MBP to function, as well as its expected behavior. To elaborate, the manifest may include the requirements of the program (i.e., system resources, dependencies, capabilities). Additionally, the manifest includes the configuration parameters affecting the program. Further, the manifest contains information about the communication channels required by the SIP, mainly for communication with other SIPs. As the manifest is declarative and machine checkable, it can be used to reason about the MBP's safety, as well as to ensure that the installation of one MBP does not break other (already-installed) MBPs. SIPs incur low overhead (relative to most commodity OSs) in terms of creation and termination as well as communication (between SIPs), primarily due to the isolation being enforced via programming language type and memory safety instead of hardware support. Singularity has been extended with sealed processes [Hunt et al. 2007].

In some cases, the reconfigurability is more subtle. For example, several isolation techniques have manual reconfigurability because the application developer specifies a portion of the application to isolate. Flicker [McCune et al. 2008] offers isolated execution environment using the late launch capability in AMD's SVM. While using hardware support, the application developer must specify the security- sensitive code for protection as the PAL. The isolated environment only lasts for executing PAL. Finally, Inktag [Hofmann et al. 2013] protects isolated HAPs from an untrusted OS. The HAP developer specifies access control policies on secure files.

Non-reconfigurable: Some of isolation policies are non-reconfigurable. That is, they are static and immutable. Non-reconfigurable policies occur when the isolation policy is inherent to the enforcement. For example, SICE [Azab et al. 2011] aims to create isolated environment for security sensitive workload, relying on a minimal TCB including only the hardware, BIOS, and the SMM. The processor saves its current state and switches to the SMM when it receives a SMI. The SMM code (i.e., the SMI handler) resides in System Management RAM (SMRAM), and SICE leverages the capability of locking the SMRAM to protect the code and data of the SMM. Once the SMRAM is initialized by the BIOS and then locked, no access is allowed except to the SMM code. The SMI handler provides memory isolation for isolated environment, and it is also able to initialize the isolated environment. Each isolated environment has a copy of security manager that prevents the isolated workload from accessing the memory of the legacy host. That is, the only policy in this case is to isolate the untrusted workload, which is non-reconfigurable as it is embedded in the enforcement mechanism.

Additionally, non-reconfigurable policies are used for the goal of enforcing least privilege on applications. For instance, some systems restrict (e.g., Jain et al. [2014]) or eliminate (e.g., Plan 9 [Pike et al. 1990]) the use of the setuid bit in order to prevent privilege escalation. Additionally, the SIP (commonly known as "rootless") feature [Apple Inc 2015] included in Mac OS X 10.11 restricts the privilege of root processes and only allows code signed by the OS vendor to perform certain privileged actions. In both these cases, the policy is non-reconfigurable, as it is a part of the enforcement mechanism. For example, in the case of SIP, the code signing enforcement will have to be turned off in the enforcement to reconfigure the policy.

Discussion: Automatic reconfigurability provides flexibility and customization for security isolation. However, it also introduces potential for configuration error that impacts the tamper-proofness of the system. For example, Linux Discretionary Access Control allows policy on subjects and objects to be reconfigured in a decentralized manner and is thus vulnerable to trojan horse and confused deputy attacks. Similarly, some of the DIFC systems discussed previously (e.g., Asbestos [Vandebogart et al. 2007] allow the policy (i.e., labels) of subjects to automatically be reconfigured based on their

communication. That is, in a mechanism similar to taint tracking, labels "float" or propagate in the direction of the data flow. While such a mechanism may seem secure as the restrictions propagate along with the data, such implicit propagation has been shown to leak information [Krohn and Tromer 2009].

On the other hand, manual reconfigurability may have the advantages of being customizable but may not be prone to the limitations of automatically reconfigurable policies. For example, a mandatory access control approach such as SELinux may provide reconfigurability but may also ensure that the reconfiguration is performed by an expert centralized administrator, reducing the potential for incorrect specification. Note that SELinux does not reconfigure policies at runtime, instead requiring a reboot. This is due to the "tranquility" property of mandatory protection systems that does not permit a change to an object's policy concurrent to its use.

Reconfigurable policies are hard to verify formally without sacrificing flexibility. For example, consider the DIFC systems discussed previously. Harris et al. [2009] use model checking to verify an application's secrecy goals and its Flume [Krohn et al. 2007] DIFC policy but cannot model all DIFC properties. Yang et al. [2011] show that verifying Flume [Krohn et al. 2007], Asbestos [Vandebogart et al. 2007], and HiStar [Zeldovich et al. 2006] is NP-hard without the use of model checking. Further, for Flume in particular, the verification problem complexity is NP-complete if security principals cannot grant or delegate privileges over their tags (i.e., less flexibility). The problem falls in P only when Flume gives up the privilege to declassify data (i.e., allow the owner to export their data contrary to the policy) as well, which severely curtails the model's flexibility but provides support for verification. Note that the verification of most policies with respect to their goals may require the program's source code if the policy is instrumented in the program.

Configurability does not imply usability, and a highly configurable and flexible policy may often be unusable for the general user. In fact, most policies governing mandatory access to low-level system resources and daemons may be considered to be non-trivial and only usable for power users. There is significant room for improvement in this area for access control in general.

Finally, the process of configuring the isolation policy may also have security implications. Requiring a human to configure the policy potentially leaves the system vulnerable until the reconfiguration occurs. Automatic reconfiguration approaches reduce the vulnerability time window, although developing software for the identified automatic reconfiguration approaches is difficult. Capabilities and IFC primitives require careful consideration of the security guarantees and may be error prone themselves. That said, enforcing a high-level policy goal (e.g., via a lattice as in DIFC approaches) provides a stronger security foundation for expressing specific policy goals. For instance, information secrecy problems arising due to data sharing between applications on Android [Nadkarni and Enck 2013], and the resultant policies to prevent such problems, can be validated using an IFC lattice.

4.3. Policy Lifetime

The lifetime of the isolation policy represents the time that the security isolation policy is being enforced. More specifically, we want to distinguish between policies that are always enforced and those that are only enforced for some subset of the duration that the isolation target (e.g., application) executes. Typically, the subset of time will begin at some point after execution begins and ends when the execution terminates. That is, isolation is on demand and stays on until the program ends. If isolation is not on demand (i.e. for some of the execution time), then it is always on (i.e., for all of the execution time). Always-on policy configuration may be performance intensive, as we discuss in the tradeoffs for this section. Hence, smart (i.e., adaptive) security isolation using on-demand policies may be necessary.

We will use the Seatbelt [Miller et al. 2012] sandboxing system used in OS X and iOS as an example to clarify the distinction between always on and on demand. A Seatbelt sandbox profile represents an isolation policy, and enforcement of that policy begins when a process calls sandbox_init() parameterized with a sandbox profile to use as a policy. Seatbelt enforcement is only activated after the process calls sandbox_init(), but this can be done in an always-on or on-demand fashion. A parent process that is not sandboxed can fork itself and cause the forked child to call sandbox_init() before calling exec() to execute a program. This method allows the sandbox to be applied in an always-on manner if the system executes all application programs with the sandbox_init() call. However, if an application were allowed to execute for some time and then called sandbox_init() on itself, then this would be an on-demand use of isolation. That is, the application would have been isolated for only part of the time that it was executing with an on-demand policy.

Additionally, as opposed to the process voluntarily isolating itself, an on-demand policy may include a monitoring phase, for example, to detect suspicious behavior that is used to trigger isolation enforcement. Hence, there may be a detection policy that is always on and an isolation policy that is enforced on demand. For example, the anomaly detection of PREC [Ho et al. 2014] is triggered only after system calls are invoked by third-party library calls to achieve low false alarms. The isolation mechanisms are then dynamically triggered after an anomaly is detected.

Always on: Most isolation policies are always on and provide isolation throughout the execution of the isolated subject. That is, the isolation policy is active when an application process starts and it remains active until the process terminates.

Some security isolation policies are always enabled in application code. For example, software-based fault isolation [Wahbe et al. 1994] sandboxes the untrusted module. Once the untrusted module is compiled, it is always sandboxed inside the fault domain. In program shepherding [Kiriansky et al. 2002], policy checks are added to malicious code such that they are triggered at all times. For example, by adding checks at the point where the system copies a basic block into code cache, it restricts program execution to only trusted code origins.

In some cases, security isolation policies are always enabled in the runtime environment. In the sealed process architecture [Hunt et al. 2007], processes are prohibited from dynamic code loading, runtime code generation, and shared memory. Sealed processes are separated with each other and only communicate through explicit mechanism. In Singularity [Hunt and Larus 2007], SIPs are isolated by both static verification and runtime checks. Gazelle [Wang et al. 2009] is a secure web browser that protects all system resources among website principals. All principals are isolated into separate protection domains while using Gazelle.

Security isolation techniques taking advantage of hardware may also be always on. In SICE [Azab et al. 2011], the workload can only enter or exit the isolated environment by triggering SMI. Before entering the isolated environment, SMI handler needs to prepare a new SMRAM layout for security manager and isolated workload. SMI handler also needs to store processor's state so it can resume the execution after the workload exists the isolated environment. Although it is possible to make SICE dynamic, workloads are made to enter the isolated environment and leave the isolated environment when they are forced to exit. Vx32 [Ford and Cox 2008] is a user-level sandbox that uses x86 protected-mode segmentation for data sandboxing and dynamic instruction translation for code sandboxing. By default, before executing guest code, Vx32 sandboxes guest data access by using the x86 processor's segmentation hardware, which loads special data segment into registers such as `ds`, `es`, and `ss`.

On demand: An isolation policy that programmatically begins enforcing isolation after some environmental change is termed *on demand*. Note that there is a subtle difference between an isolation policy's lifetime characteristic of "on demand" and its configurability characteristic of "automatically reconfigurable." In many cases, a policy may be both. However, semantically, the characteristics differ. For example, a non-reconfigurable or manually reconfigurable policy may be triggered after some suspicious activity is detected.

On-demand policies can be as simple as turning the policy on (and potentially off) depending on the context. More complex security isolation policies can continuously change the degree of security isolation depending on system context. However, developers must predefine when security isolation should be active or how it should adapt to the system's context. There is also a danger of security violation occurring while the security isolation policy is relaxed or not active.

One option for on-demand security isolation is to allow developers to request security isolation for specific sections of code. Isolating sections of code removes the overhead penalty of security isolation for sections that do not need it. This technique also allows the assignment of different security isolation policies for multiple sections of the same application. Capsicum [Watson et al. 2010] enforces developer specified security isolation after the process calls cap_enter (a Capsicum provided function). This function and other modifications are added into a process's code by developers to specify when it should be isolated. After a process calls cap_enter, it enters an isolated environment called capability mode. In capability mode, a process is granted privileges in the form of capabilities that are also determined by developers. This self-compartmentalization ability is especially useful when a trusted program isolates itself before processing untrusted input. If the input caused malicious behavior (e.g., buffer overflow), then the program would be isolated at this point and exploit's effects can be confined.

A security isolation system can vary isolation strength in response to subject behavior. As an application starts to behave suspiciously, it will be contained under a proper security isolation mechanism. For example, PREC [Ho et al. 2014] provides this type of security isolation lifetime for Android applications. As soon as an application makes a suspicious system call sequence, the execution of those suspicious system calls will be delayed. These delays increase exponentially for each additional malicious activity. Eventually, the delays are so severe that malicious activity becomes ineffective. This technique is especially useful when defending against attacks that abuse concurrency bugs such as race conditions. The delays can disrupt the delicate timing required to trigger these concurrency bugs. Normal sequences of system calls cause decreases in the delay time, allowing the process to regain normal speeds.

Discussion: On-demand policies such as Capsicum [Watson et al. 2010] may improve performance by only incurring the overhead of isolation enforcement for a subset of execution time. However, there may also be performance overhead involved in activating or deactivating the isolation enforcement. If this overhead is large, and if the on-demand policy is frequently turned on or off, then the overall performance impact may be even more than an always-on policy. Additionally, evaluating performance for an on-demand isolation system is difficult. That is, the on-demand policy is dependent on the runtime context, and an objective evaluation of the runtime context may involve a number of factors that may be hard to simulate realistically, such as the kind of application (e.g., text editor versus game), aspects of the user scenario (e.g., user-initiated access, scenario-specific storage access), and so on. Additionally, by causing suspicious processes to be delayed, PREC may grant performance gains to other processes running on the same system. However, the performance overhead of a benign process experiencing delays due to false positives may be severe.

On-demand policies may also require source code access, especially if the policy is to be triggered by the application itself. For instance, some DIFC systems (e.g., HiStar [Zeldovich et al. 2006]) require applications to explicitly change their labels to read/write data. Note that in DIFC systems, while the overarching secrecy and integrity label checks are always on, a subject may change its own isolation policy based on its requirements and capabilities. Further, adding the cap_enter sytem call to a program to enable on-demand isolation via Capsicum requires access to source code as well. However, it may be possible to reverse engineer and rewrite binaries to introduce a cap_enter system call as well. Some solutions that do not require the policy to be triggered in application code, such as PREC, do not have any code requirements and can be applied to unmodified applications.

In contrast, always-on policies are often simple and easier to verify. On the contrary, on-demand isolation policies are difficult to verify due to their dynamic nature. In addition to verifying that the isolation policy is configured correctly, developers of on-demand policies must also verify that the policy is enforced at the right time. The decision of when to use cap_enter in a program is important, and putting it in the wrong location may allow a process to become compromised before it is isolated. However, systems like PREC that tolerate false positives may reduce the cost of a poorly configured policy that was not verified.

4.4. Possible Future Work

As mentioned in Section 3 and previous parts of Section 4, most existing security isolation techniques are static. That is, the isolation mechanisms and policies cannot be dynamically configured during runtime to adapt to application or environment changes. However, there are scenarios where the system and users would benefit from a system that adaptively configures its isolation mechanisms or policies. For example, consider a scenario where an application initially runs on a local dedicated data center. When the application workload increases, the resource requirement may exceed the capacity of the local data center. To accommodate the transient overload condition, the application changes its configuration by offloading some tasks into a public cloud. The security isolation must adapt to this environmental change. For example, the isolation granularity must change from the whole application to individual tasks. While one approach is to begin with individual task isolation, it would unnecessarily raise the isolation cost during normal workloads. However, it is a challenging task to develop such an adaptive security isolation system, which is part of our future work and beyond the scope of this survey article.

5. CONCLUSION

In this article, we performed a study of the different techniques used in security isolation and proposed a hierarchical classification system. At the top were two broad aspects: mechanism and policy. Each aspect was broken into dimensions. For mechanism, we considered dimensions of enforcement location and isolation granularity. For policy, we considered dimensions of generation, configurability, and lifetime. Each dimension was then classified via its categories and subcategories. We applied the classification system to a representative set of research articles that represent a diverse set of isolation techniques. We then discussed the tradeoffs of different technique categories based on performance overhead, code requirements, and security assurance.

ACKNOWLEDGMENTS

Any opinions, conclusions, or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Anurag Acharya and Mandar Raje. 2000. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th Conference on USENIX Security Symposium-Volume 9*. USENIX Association, 1–1.

AMD64. 2005. Secure virtual machine architecture reference manual. *AMD Publication* 33047 (2005).

Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. 2007. Libra: A library operating system for a jvm in a virtualized execution environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*. ACM, 44–54.

J. P. Anderson. 1972. *Computer Security Technology Planning Study*. ESDTR-73-51. Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA. (Also available as Vol. I, DITCAD-758206. Vol. II DITCAD-772806).

Apple Inc. 2015. System Integrity Protection Guide. Retrieved from https://developer.apple.com/library/mac/documentation/Security/Conceptual/System_Integrity_Protection_Guide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40016462-CH1-DontLinkElementID_15.

Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. 2011. Sice: A hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, 375–388.

Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged app sandboxing for stock android. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*. 691–706.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS Operat. Syst. Rev.* 37, 5 (2003), 164–177.

Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. 2013. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 239–252.

Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding applications from an untrusted cloud with haven. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 41–46.

Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The turtles project: Design and implementation of nested virtualization. In *OSDI*, Vol. 10. 423–436.

Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. 1995. *Extensibility Safety and Performance in the SPIN Operating System*. Vol. 29. ACM.

Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. 1989. The modula–3 type system. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 202–212.

Benjie Chen and Robert Morris. 2003. Certifying program execution with secure processors. In *HotOS*. 133–138.

Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang, and Wenbo Mao. 2007. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor.

Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. 2008. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGOPS Operating Systems Review*, Vol. 42. ACM, 2–13.

Yueqiang Cheng, Xuhua Ding, and R. Deng. 2013. Appshield: Protecting applications against untrusted operating system. *Singaport Management University Technical Report, SMU-SIS-13* 101 (2013).

Chris Clayton. 2013. Understanding Application Domains. Retrieved from https://blogs.msdn.microsoft.com/cclayton/2013/05/21/understanding-application-domains/. (2013).

Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. 2011. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, 189–202.

Fernando J. Corbató and Victor A. Vyssotsky. 1965. Introduction and overview of the Multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*. ACM, 185–196.

Jonathan Corbet. 2009. Seccomp and sandboxing. LWN.net, May (2009).

Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, Christine Paulin-Mohring, César Munoz, Chetan Murthy, Catherine Parent, Amokrane Saibi, and others. 1995. *The Coq Proof Assistant Reference Manual, Version 5.10*. Technical Report. INRIA, France. Research Report, RT-0177, inria-00069994.

K. Crary, Neal Glew, Dan Grossman, Richard Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. 1999. TALx86: A realistic typed assembly language. In *Proeedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*. 25–35.

Dorothey E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (May 1976).

Jeff Dike and others. 2001. User mode linux. (2001).

Alan M. Dunn, Michael Z. Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. 2012. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Presented as Part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 61–75.

William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. Understanding android security. *IEEE Sec. Priv.* 1 (2009), 50–57.

Dawson R. Engler, M. Frans Kaashoek, and others. 1995. *Exokernel: An Operating System Architecture for Application-level Resource Management*. Vol. 29. ACM.

Ulfar Erlingsson. 2003. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. Technical Report. Cornell University.

Bryan Ford and Russ Cox. 2008. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*. Boston, MA, 293–306.

Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. 1996. Microkernels meet recursive virtual machines. In *OSDI*, Vol. 96. 137–151.

Bill Frantz. 1988. KeyKOS-asecure, high-performanceenvironmentforS/370. In *Proc. of SHARE* 70 (1988), 465–471.

Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. 2003. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 193–206.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazieres, John C. Mitchell, and Alejandro Russo. 2012. Hails: Protecting data privacy in untrusted web applications. In *OSDI*. 47–60.

William R. Harris, Somesh Jha, and Thomas Reps. 2010. DIFC programs by automatic instrumentation. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM, 284–296.

William R. Harris, Nicholas A. Kidd, Sagar Chaki, Somesh Jha, and Thomas Reps. 2009. Verifying information flow control over unbounded processes. In *FM 2009: Formal Methods*. Springer, 773–789.

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 165–181.

Matt Helsley. 2009. LXC: Linux container tools. IBM devloperWorks Technical Library (2009).

Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick Drew McDaniel. 2007. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*, Vol. 7. 34.

Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. 2014. PREC: Practical root exploit containment for android devices. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. ACM, 187–198.

Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. Inktag: Secure applications on an untrusted operating system. *ACM SIGPLAN Not.* 48, 4 (2013), 265–278.

Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Trans. Softw. Eng.* 23, 5 (1997), 279–295.

Jon Howell, Bryan Parno, and John R. Douceur. 2013. Embassies: Radically refactoring the web. In *NSDI*. 529–545.

Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. 2007. Sealing OS processes to improve dependability and safety. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 341–354.

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

Intel. 2007. Intel Trusted Execution Technology. Retrieved from http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html. (2007).

Bhushan Jain, Chia-Che Tsai, Jitin John, and Donald E. Porter. 2014. Practical techniques to obviate setuid-to-root binaries. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM, New York, NY,, Article 8, 14 pages. DOI:http://dx.doi.org/10.1145/2592798.2592811

Suman Jana, Donald E. Porter, and Vitaly Shmatikov. 2011. TxBox: Building secure, efficient sandboxes with system transactions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP)*. IEEE, 329–344.

Håvard D. Johansen, Eleanor Birrell, Robbert van Renesse, Fred B. Schneider, Magnus Stenhaug, and Dag Johansen. 2015. Enforcing privacy policies with meta-code. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys'15)*. Article 16. DOI:http://dx.doi.org/10.1145/2797022.2797040

Poul-Henning Kamp and Robert N. M. Watson. 2000. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, Vol. 43. 116.

Taesoo Kim and Nickolai Zeldovich. 2013. Practical and effective sandboxing for non-root users. In *Presented as Part of the 2013 USENIX Annual Technical Conference*. USENIX, 139–144.

Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2002. Secure execution via program shepherding. In *Proceedings of the USENIX Security Symposium*, Vol. 92.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, and others. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 207–220.

Kirill Kolyshkin. 2006. Virtualization in linux. *White Paper, OpenVZ* (2006).

Maxwell Krohn and Eran Tromer. 2009. Noninterference for a practical DIFC-based operating system. In *Proceedings of the IEEE Symposium on Security and Privacy*. 61–76.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. 321–334.

Yanlin Li, J. M. McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. 2014. MiniBox: A two-way sandbox for x86 native code. In *Proceedings of the 2014 USENIX Annual Technical Conference*.

Jed Liu, Michael D. George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 321–334.

Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 81–96.

Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 461–472.

Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC architecture. In *Usenix Security*. 15.

Steven McCanne and Van Jacobson. 1993. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX Association, 2–2.

Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*. IEEE, 143–158.

Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. 2007. Minimal TCB code execution. In *Proceedings of the IEEE Symposium on Security and Privacy, 2007. SP'07*. IEEE, 267–272.

Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. In *ACM SIGOPS Operating Systems Review*, Vol. 42. ACM, 315–328.

Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 1–1.

Paul Menage. 2004. Control Groups. Retrieved from https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt. (2004).

Dirk Merkel. 2014. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.* 2014, 239 (2014), 2.

James Mickens. 2014. Pivot: Fast, synchronous mashup isolation using generator chains. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, 261–275.

Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philip Weinmann. 2012. *iOS Hacker's Handbook*. John Wiley & Sons.

Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. *Safe Active Content in Sanitized JavaScript*. Technical Report. Google, Inc.

MSDN. 2012. Understanding Enhanced Protection Mode. Retrieved from http://blogs.msdn.com/b/ieinternals/archive/2012/03/23/understanding-ie10-enhanced-protected-mode-network-security-addons-cookies-metro-desktop.aspx. (2012).

Adwait Nadkarni and William Enck. 2013. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. 1029–1042.

National Security Agency. 2009. Security-Enhanced Linux (SELinux). Retrieved from http://www.nsa.gov/research/selinux. (2009).

George C. Necula. 1998. *Compiling with Proofs*. Technical Report. DTIC Document.

George C. Necula and Peter Lee. 1998. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*. 333–344.

David M. Nicol, William H. Sanders, William L. Scherlis, and Laurie A. Williams. 2012. Science of Security Hard Problems: A Lablet Perspective. Science of Security Virtual Organization Web. (Nov. 2012).

Kaan Onarlioglu, Collin Mulliner, William Robertson, and Engin Kirda. 2013. Privexec: Private execution as an operating system service. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*. IEEE, 206–220.

Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, and Tim Stack. 2003. Upgrading transport protocols using untrusted mobile code. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 1–14.

Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM, 71–72.

Phu H. Phung and Lieven Desmet. 2012. A two-tier sandbox architecture for untrusted JavaScript. In *Proceedings of the Workshop on JavaScript Tools*. ACM, 1–10.

R. Pike, D. Presotto, K. Thompson, and H. Trickey. 1990. Plan 9 from bell labs. In *Proceedings of the UKUUG Conference*. London, UK, 1–9.

Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. 2011. ADsafety: Type-based verification of JavaScript sandboxing. In *Proceedings of the 20th USENIX Conference on Security*. Usenix Association, 12–12.

Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the library OS from the top down. *ACM SIGPLAN Not.* 46, 3 (2011), 291–304.

Shaya Potter, Jason Nieh, and Matt Selsky. 2007. Secure isolation of untrusted legacy applications. In *LISA*, Vol. 7. 1–14.

Niels Provos. 2003. Improving host security with system call policies. In *USENIX Security*, Vol. 3.

Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, and Richard D. Schlichting. 2006. System call monitoring using authenticated system calls. *IEEE Trans. Depend. Sec. Comput.* 3, 3 (2006), 216–229.

Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 199–212.

Dan Rosenberg. 2014. Qsee trustzone kernel integer over flow vulnerability. In *Black Hat Conference*.

Joanna Rutkowska. 2012. Introducing Qubes 1.0. Retrieved from http://theinvisiblethings.blogspot.com/2012/09/introducing-qubes-10.html. (2012).

Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Van Doorn, John Linwood Griffin, Stefan Berger, Reiner Sailer, Enriquillo Valdez, Trent Jaeger, and others. 2005. sHype: Secure hypervisor approach to trusted virtualized systems. Technical Report RC23511 (2005).

Jerry Saltzer and Mike Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (Sep. 1975).

Nuno Santos, Rodrigo Rodrigues, and Bryan Ford. 2012. Enhancing the OS against security threats in system administration. In *Middleware 2012*. Springer, 415–435.

Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (Feb. 2000), 30–50. DOI: http://dx.doi.org/10.1145/353323.353382

Fred B. Schneider, Greg Morrisett, and Robert Harper. 2001. A language-based approach to security. In *Informatics*. Springer, 86–101.

Zhiyong Shan, Xin Wang, Tzi-cker Chiueh, and Xiaofeng Meng. 2012. Facilitating inter-application interactions for os-level virtualization. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 75–86.

Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*. ACM, New York, NY, 170–185. DOI: http://dx.doi.org/10.1145/319151.319163

Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*. 553–567.

Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. 2006. Reducing TCB complexity for security-sensitive applications: Three case studies. *ACM SIGOPS Operat. Syst. Rev.* 40, 4 (2006), 161–174.

Christopher Small. 1997. A tool for constructing safe extensible C++ systems. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*. 175–184.

Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 275–287.

Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazieres. 2012. Addressing covert termination and timing channels in concurrent information flow systems. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 201–214.

Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. 2014. Protecting users by confining JavaScript with COWL. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

Marc Stiegler, Alan H. Karp, Ka-Ping Yee, Tyler Close, and Mark S. Miller. 2006. Polaris: Virus-safe computing for windows XP. *Commun. ACM* 49, 9 (2006), 83–88.

Richard Ta-Min, Lionel Litty, and David Lie. 2006. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 279–292.

Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. 2010. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the USENIX Security Symposium*. 371–388.

Trusted Computing Group. 2011. TPM Main Specification. Retrieved from http://www.trustedcomputinggroup.org/resources/tpm_main_specification. (2011).

Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 9th European Conference on Computer Systems*. ACM, 9.

Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. 2007. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.* 25, 4 (December 2007).

Thorsten Von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. 1999. J-kernel: A capability-based operating system for java. In *Secure Internet Programming*. Springer, 369–393.

David A. Wagner. 1999. *Janus: An Approach for Confinement of Untrusted Applications*. Ph.D. Dissertation. Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1994. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, Vol. 27. ACM, 203–216.

Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. 2009. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the USENIX Security Symposium*, Vol. 28.

Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 33–47.

Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical capabilities for UNIX. In *Proceedings of the USENIX Security Symposium*. 29–46.

Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. 2002. Scale and performance in the Denali isolation kernel. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 195–209.

Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. 2014. Airbag: Boosting smartphone resistance to malware infection. In *Proceedings of the Network and Distributed System Security Symposium*.

Weiyi Wu and Bryan Ford. 2015. Deterministically deterring timing attacks in Deterland. *Conference on Timely Results in Operating Systems (TRIOS)*.

Yongzheng Wu, Sai Sathyanarayan, Roland H. C. Yap, and Zhenkai Liang. 2012. Codejail: Application-transparent isolation of libraries with tight program interactions. In *Computer Security–ESORICS 2012*. Springer, 859–876.

Xi Xiong, Donghai Tian, and Peng Liu. 2011. Practical protection of kernel integrity for commodity OS from untrusted extensions. In *NDSS*.

Rubin Xu, Hassen Saïdi, and Ross Anderson. 2012. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium*. 539–552.

Jisoo Yang and Kang G. Shin. 2008. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 71–80.

Zhi Yang, Lihua Yin, Miyi Duan, and Shuyuan Jin. 2011. Poster: Towards formal verification of DIFC policies. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, New York, NY, 873–876. DOI:http://dx.doi.org/10.1145/2046707.2093515

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. IEEE, 79–93.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. 263–278.

Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. 2011. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP)*. IEEE, 313–328.

Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. 2011. ARMor: Fully verified software fault isolation. In *Proceedings of the 2011 International Conference on Embedded Software (EMSOFT)*. IEEE, 289–298.

Xin Zhao, Kevin Borders, and Atul Prakash. 2005. Svgrid: A secure virtual environment for untrusted grid applications. In *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*. ACM, 1–6.

Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014a. ARMlock: Hardware-based fault isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 558–569.

Zongwei Zhou, Miao Yu, and Virgil D. Gligor. 2014b. Dancing with giants: Wimpy kernels for on-demand isolated I/O. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, 308–323.

Received September 2015; revised July 2016; accepted August 2016