TWC: Small: XCap: Practical Capabilities and Least Authority for Virtualized Environments

Despite a number of radical changes in how computer systems are used, the security model of a modern operating system is based on principles laid down by the first time-sharing computing environments four decades ago. Early computer systems were designed to fulfill a simple security goal: provide isolation of multiple users in a time-sharing environment. Security attacks against these systems were a pastime for small communities of hobbyists. Today, systems with essentially the same security model are required to operate in the face of targeted security attacks sponsored by a multi-national malware economy, commercial espionage, and government intelligence agencies. A lack of strong isolation between processes and users, an outdated access control model, excessive authority granted to applications, and a shared operating system kernel make it challenging to secure modern systems against these sophisticated attacks.

Proposed Research We propose creating *XCap*, a secure environment for least-authority execution of applications and system services. Unmodified, untrusted, off-the-shelf applications, running on untrusted operating systems, will be isolated by a virtual machine manager. XCap builds on two principles: *strong isolation* and *secure collaboration.* XCap's default, a *share nothing* environment, will be augmented by a capability access control model—a clean and general abstraction, enabling fine-grained delegation of rights in a flexible and manageable way. In XCap, capabilities will serve as a general foundation for constructing least privilege services out of existing components of the traditional operating system stack. Furthermore, capabilities enable formal reasoning about authority of individual applications, and the system overall.

Intellectual Merit Our first contribution will be the design of a practical capability system that is fine-grained enough to enforce useful policies while being efficient enough to avoid getting in the way of application execution. Our second main contribution will be a collection of decomposed system services that runs under XCap, permitting, for example, fine-grained protection of file accesses by VMs. Several principles constitute the core of XCap's architecture. First, XCap relies on a hardware-level virtualization platform to provide strong isolation of individual applications. Each application runs in a fresh, private copy of an operating system, communicating with the rest of the system through a minimal, capability-mediated interface. Second, XCap builds on a capability access control model. Capabilities explicitly name all resources in the system, and provide the only way of accessing them. Third, XCap maximizes the principle of least authority—XCap redesigns common operating system services in such a way that the authority of individual applications and services is minimized. Each component possesses the smallest subset of rights required to accomplish its task, e.g. a shared file system has rights to provide a naming service, but no rights to access the content of the files. Thus, the effect of the compromise of an individual application is restricted to the set of resources that the application can access.

Broader Impacts We expect that the proposed work will provide a foundation for mitigating the vast economic damage that is enabled by the security architecture of our current software systems. By running untrusted applications in fresh virtual machines, and by destroying these VMs when their applications are no longer needed, the kind of sophisticated multi-stage attacks that are the basis of today's botnets and related activities will be largely eliminated. XCap will be open source, directly benefiting the broader community.

Key Words. security; capability access control; virtualization; least privilege; strong isolation.

Contents

1 Overview

Modern software systems inherit their architecture, software development methodology, and security model from time-sharing operating systems developed four decades ago. Desktop, server, cloud, and even industrial control systems rely on a large stack of commercial off-the-shelf applications, which run on top of a monolithic operating system kernel. Each application runs with the full set of privileges of the user, has access to the entire file and application space of the user, and can access the complete interface of a complex operating system kernel, and a number of privileged systems components.

We believe the security model exposed by existing software systems is fundamentally too weak; it fails to provide adequate isolation between computations. Today, a single compromised application opens the door to multi-stage exploits that all too often end up giving an attacker full control over a machine or network of machines. A typical drive-by download attack [83] exploits one of the vulnerabilities in a semantically rich web browser interface. It then gains administrative privileges through exploiting an even larger interface of the operating system. Subsequently, it registers itself as an early stage boot module in order to retain control over the system across reboots [15, 42, 45]. The attack code transparently injects logging components into commonly used applications, collects sensitive financial information and user credentials, and conceals itself from the user and anti-virus applications by creating a thin layer of virtualization inside the operating system kernel. The entire attack is transparent to the user.

Despite a number of advances in program analysis, software testing, and verification, it is unlikely that the large, semantically rich interfaces exposed by operating systems, browsers, and other pieces of modern infrastructure can be effectively secured. On the other hand, even in the face of persistent exploitable vulnerabilities, it is possible to confine the effects of individual attacks by partitioning large untrusted system components into pieces that are isolated by a small software layer that uses a formal access control model that explicitly defines interactions between untrusted components and outcomes of untrusted computations.

1.1 XCap: Strong Isolation and Least Authority

The proposed work, *XCap*, will provide a practical environment for enforcing the principle of least authority for off-the-shelf UNIX applications. The XCap execution environment will enforce strong isolation between applications, each of which will be invoked with a minimal set of privileges, ensuring that successful attacks against application logic are confined to a single instance of a single application, as opposed to being persistent and spreading to other system components. The basis for XCap's design is (1) applications are isolated using a virtual machine monitor and (2) all sharing between virtual machines is mediated by a capability access control model that explicitly describes the external resources available to the application and also the ways in which the application can interact with the rest of the system (Figure 1). In more detail, XCap builds on the following principles:

Strong isolation To isolate applications and components implementing system services, XCap relies on hardware-level, full-system virtualization. The interface exported by a hypervisor—the x86 instruction set—has far simpler semantics than, for example, the system call interface exported by an operating system. This interface supports strong and secure isolation of all relevant resources. Furthermore, although current VMMs may contain their own exploitable vulnerabilities, recent

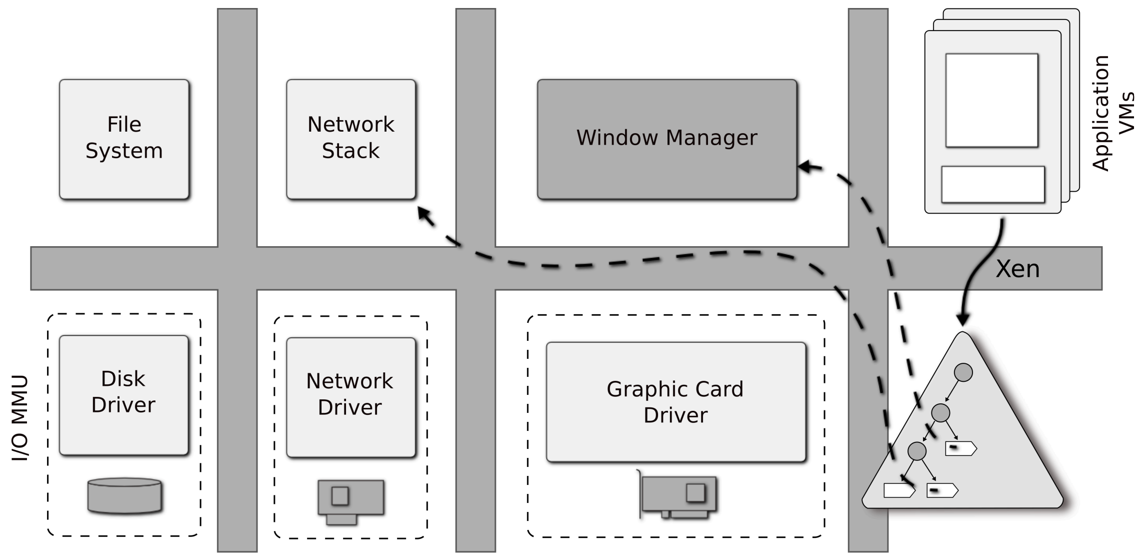

Figure 1: XCap architecture. Applications, device drivers, and operating system services are strongly isolated inside the Xen virtual machine. Each VM runs a fresh, private copy of the operating system kernel. A trusted window manager assembles the screens of individual virtual machines in a single windowed environment, so disaggregation is transparent to the user. Capabilities explicitly describe the external resources available to each virtual machine.

advances in formal verification [3,33,50,63,73,115,123] lead us to believe that within the foreseeable future, fully verified hypervisors will be available. The proposed work, however, will be based on Xen: a full-featured, open-source, de-facto industry standard hypervisor.

No default sharing XCap will provide an execution environment where each application is started in a fresh virtual machine instance with a newly-booted operating system. No resources will be shared between VMs by default, and even access to the hypervisor will be mediated by the XCap capability layer.

Capability access control To reason about authority of applications in presence of a dynamic collaboration and dynamic exchange of privileges, XCap will rely upon a capability access control model. Capabilities explicitly name all resources of the system and provide the only way to access them. To provide a general mechanism for building secure applications, the capability model in XCap will be available at all levels of the software stack. For example, a web wiki application will create file objects and protect them with capabilities. However the set of core objects, e.g. memory objects and communication primitives, will be protected by the hypervisor. This allows XCap to reason about coarse-grained confinement of applications without trusting the application level components.

Practical least authority A major part of the proposed research is creating a collection of disaggregated operating system services: file systems, network stack, device drivers, etc., according to the principle of least authority. These services will, for the most part, be decomposed along existing interfaces. On the other hand, some services such as the filesystem will require significant reengineering in order to implement naming, buffering, and metadata services—all without giving the filesystem code access to the contents of the files it is managing. Thus, in XCap, an exploitable bug in the filesystem can lead to a local lack of availability—a denial of service attack—but will not violate confidentiality and integrity of data. The proposed research, if successful, will produce disaggregated versions of several widely used system services, and will also result in a collection of best practices for disaggregation of other services and applications.

Patterns for secure collaboration XCap is intended to create a set of mechanisms and techniques for implementing secure collaboration across mutually untrusted applications. The following mechanisms are important.

API proxying: XCap will support a set of proxy components aimed at selectively disabling API, monitoring, logging, and accounting access to trusted and untrusted components. A file proxy can be used to enforce read-only access or file versioning. A file system proxy can enforce access control decisions in front of an untrusted file system.

Secure synchronization primitives: To guarantee availability in the presence of mutually untrusted components XCap will implement a set of secure synchronization primitives, ensuring that a single compromised component cannot block access to a shared resource.

Sealed compartments: XCap will provide mechanisms for creating "islands" of isolation by sealing a complex computation, which may span multiple virtual machines, in a capability-enforced compartment. Sealing supports waiting for the result of the untrusted computation and receiving it through a well-structured interface. Sealing will be implemented using the capability system, which guarantees confinement of authority. Sealing provides a safe way to execute untrusted applications.

Optimizations XCap is intended to be practical and will exploit a modern high-performance hypervisor. To address overheads of fine-grained virtual machine isolation, XCap will use copy-on-write sharing of VM's memory, lightweight VM cloning, and fast inter-VM communication mechanisms. To support applications which manipulate huge numbers of capabilities, XCap will provide fast capability caches that can implement capability checks using a small number of memory references.

1.2 XCap Examples

Figure 2 shows how XCap might decompose a collection of desktop applications and system services. Assume the user opens a PDF file. XCap creates a new virtual machine with its own OS, a private copy-on-write file system, and a PDF reader application. When the PDF reader is no longer needed, the VM will be shut down and all changes to the file system will be discarded.

To communicate with the outside world, the VM for the PDF reader is given a collection of capabilities permitting it to (1) display a window that interacts with the user and (2) read (and perhaps write) the PDF file. The PDF is exported to the VM through a special file proxy-stub inter-VM communication mechanism in a way which is transparent to the PDF reader application.

Attack 1 Assume that the PDF reader contains a vulnerability, perhaps in its JavaScript engine, that is exploited by a malicious segment of the PDF document that the user is trying to view. Vulnerabilities of this kind are very common [87]. Following the attack, the PDF reader is running remote code and will probably launch a local privilege escalation attack in order to gain root-level privileges. In XCap, the OS kernel supporting the PDF reader is not specifically hardened, and so the second stage of this attack will probably succeed. However, the now-hostile VM is isolated from the rest of the system and it will not be able to accomplish any addition mischievousness. The only

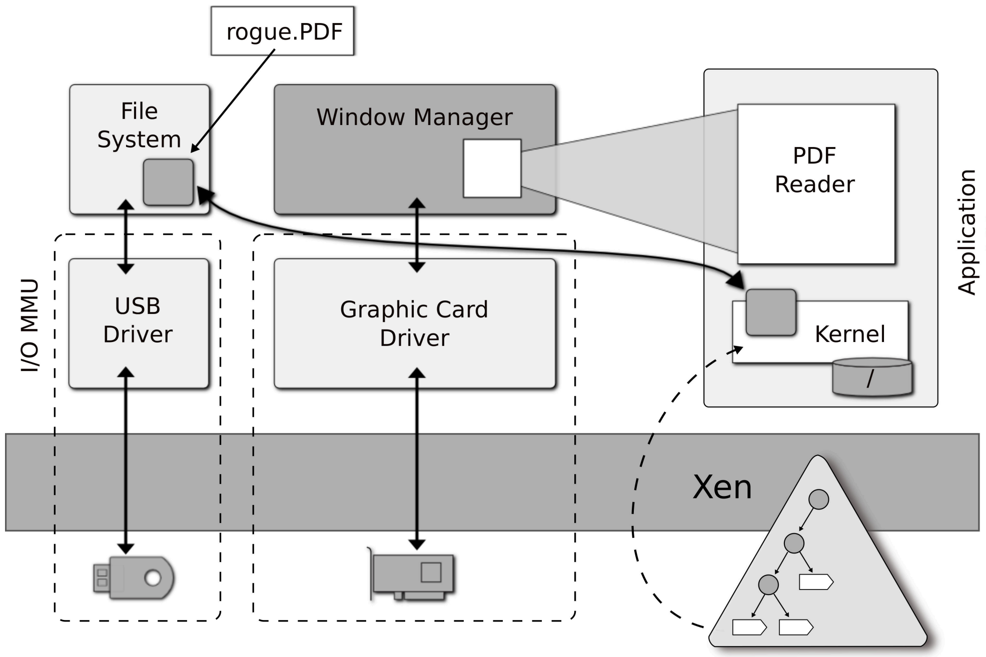

Figure 2: Example of isolating an untrusted PDF Reader application.

avenues for egress are the window seen by the user and the file system proxy stub, both of which we believe are simple enough that they can be validated through testing and even formal verification.

Attack 2 Assume that the PDF file loaded by the user is on a USB drive. XCap will spawn a fresh VM with capabilities for accessing the USB devices and for exporting a file. Again, no specific actions have been taken to harden the OS running in the USB VM and a vulnerability [81] may permit it to be compromised. The hostile OS will be able to modify or delete files on the USB drive, and it will be able to export a file of its own choosing, but further malicious effects will not be possible, and once the file has been exported the VM will be destroyed.

1.3 Why Harden Virtual Machines?

A large amount of effort has been put into hardening the system call interfaces exported by operating systems such as Linux, Windows, and MacOS. Considerable progress has been made. However, the basic design of these interfaces—they have rich semantics and a large number of system calls—means that they are an ongoing source of novel vulnerabilities and there does not appear to be any end in sight. We believe that the narrower, lower-level interface exposed by a hypervisor is a much better choice for enforcing security guarantees. By invoking each device driver and each application in a fresh VM, malicious effects can be contained without making too many difficult changes to the huge mass of legacy code. The following properties of VM-level isolation are key.

No incidental sharing Even though some kinds of incidental sharing can be stopped (using, for example, quotas for file usage, inodes, memory, number of processes, and CPU usage) it remains easy to write a UNIX program that effects a local denial-of-service attack on all other processes running on the same OS kernel. Implementing isolation at the VM boundary is more robust, eliminating all of these sources of sharing at once. If an XCap application needs to share resources with another VM, it requests permissions explicitly using the capability model.

Smaller TCB General-purpose operating systems have essentially become large software libraries that execute monolithically at the maximum privilege level. The interface and code base of a hypervisor are relatively small and they are suitable targets for full formal verification [63].

Simple access control The XCap access control model inherits the relatively simple semantics of the virtual machine interface. In contrast to the operating system level access control enforcement frameworks such as SELinux [108], XCap does not have to enforce isolation of the operating system components at the level of access control rules. For the majority of desktop and server applications, e.g. PDF viewers, web browsers, email clients, web servers, the access control model consists of a single rule granting access to a single file or folder.

1.4 Why Capabilities?

Capabilities [34] are a flexible, powerful mechanism for constructing secure systems. Capabilities enable construction of environments with a completely isolated or controlled information flow. The *take-grant* version of the capability model has been shown to be *safe*: by looking at the initial distribution of capabilities, it is possible to decide the upper limit on authority and information flow across individual subjects in face of all possible transformations to the model and exchange of rights [66]. Capabilities are a suitable engineering choice for constructing isolated systems according to the principle of least privilege [97]. Finally, the capability model provides a natural way to model authority in the dynamic environment with a large number of principals. This is a perfect match for the XCap environment, which encourages fine-grained principals to minimize authority of individual activities and computations.

2 Threat Model

The core assumptions behind XCap are that (1) we can secure a hypervisor in addition to a capability enforcement system and (2) we cannot effectively secure application logic, system services, and device drivers. Thus, our threat model involves attacks that exploit logic errors (buffer overflows, integer overflows, faulty use of encryption libraries, etc.) in applications, system services, and device drivers. These errors can manifest in a variety of ways including compromised applications, local privilege escalation attacks against the operating system, and compromised OS kernels due to device driver vulnerabilities.

Not in the threat model The proposed work does not address hypervisor vulnerabilities, but rather we speculate that fully formally verified hypervisors will become available in the foreseeable future. We also do not plan to address side-channel attacks; while these are important, addressing them is simply beyond the scope of the proposed work.

3 The Landscape of Attacks and Defenses

Over the last decade, attacks on computer systems have undergone major changes in their exploit discovery tools, attack complexity, and targeted parts of the system. Today, sophisticated attacks can exploit 6-10 different vulnerabilities, in a chain of up to 10 different stages [91, 92]. Attackers routinely use fuzzing tools [78, 80], and attack every possible layer of computer system: application, operating system, device drivers, operating system services (e.g. network stack, stack of USB protocols, file systems), virtual machine monitor, hardware devices, CPU system management mode, and even on-chip Intel Active Management Technology [2]. XCap is designed under a pragmatic assumption that any application, operating system service, and even hardware component of the computer system can be attacked, and can be used as an intermediate stage to compromise the rest of the system.

3.1 Attacks on the Modern Computer Systems

Attacks on the shared kernel—local privilege escalation: Despite the fact that a number of static [8, 31, 39] and dynamic (stack guards [32], address space randomization [60], executable space protection [110], control flow integrity [1, 40], code integrity checks [103]) mechanisms have been invented to protect execution of the operating system kernel, these kernels remain vulnerable [24]. Attackers come up with new ways to bypass these protection techniques [60, 88, 93, 104]. Taking into account the rate at which modern operating systems grow in size, monolithic kernels will likely remain vulnerable [86].

Being unable to prevent kernel exploits, XCap takes a pragmatic security approach and blocks attacks on the operating system kernel at the level of the virtual machine boundary. XCap runs each application in a separate virtual machine, with its own private copy of the operating system kernel. Even if the kernel is attacked, the attacker can only gain access to the resources that are available to the application.

Attacks on device drivers: A recent vulnerability study of the Linux kernel found that 20% of the vulnerabilities in the Linux kernel are in the device drivers [24]. Both Windows and Linux device drivers are notorious for being sources of software errors [27, 47, 49]. Attacks routinely exploit vulnerabilities in every part of the device driver stack: the "autorun" protocol [64, 65], device partition table handling code, which is read by the operating system even if the autorun feature is disabled [89], the USB Host Controller [12], and the device drivers themselves [23, 81]. Stuxnet [42] used the LNK vulnerability, which allowed it to load an arbitrary DLL just by browsing to a folder in Windows Explorer [67].

XCap relies on the virtualization technology to securely isolate individual device drivers in separate virtual machines. Similar to applications, each device driver runs its own private copy of the operating system kernel, and has a well-defined, capability-mediated channel to the rest of the system. XCap extends device isolation layer with the careful use of the I/O MMU mechanisms [4, 98].

Attacks on the device hardware: In recent years, multiple attacks on device hardware have been demonstrated including attacks on network card firmware [68, 116], USB keyboards [25], a "Ring -3" attack on the Intel Active Management Technology [2], and even the I/O MMU engine [99]. A number of attacks can be launched via the DMA engine of the computer system (see [100] for a good overview). Attacks abuse the USB On-The-Go (OTG) [69] and FireWire [6] controllers [9, 14, 19, 71], exploit the firmware of the controller [35], manufacture a malicious controller [10], or just drive a USB controller from a Linux-based cell phone [5]. Attacks have targeted both the main memory of a system [98] and peer DMA devices [43, 101].

To protect the system, XCap combines device driver isolation with I/O MMU and DMA protection techniques [99, 100, 125]. To ensure atomicity of the I/O MMU protection [113], XCap relies on the late launch mechanism to protect itself, and the memory of the device drivers.

Application attacks: The so-called "drive by download" attack [83] is an example of an attack on the web browser application, and has become one of the most common ways to spread malware on the web [90]. The attack transparently embeds a link to a malicious Javascript script into a seemingly safe web page. When opened, the code exploits a vulnerability in the user's web browser. Provos et al. discovered that 1.3% of all Google search results contain a drive-by-download attack [83].

XCap addresses application attacks by securely isolating applications inside independent virtual machines, and by applying the principle of least authority to those virtual machines. Each application is started with a minimal set of privileges required to accomplish its task. The capability access control model along with proxy components, will limit the application VM to accessing a small set of configuration files, in a read-only manner.

3.2 The State of the Art is Inadequate

Traditional operating systems It is our claim that all aspects of existing operating systems: architecture, abstractions, isolation mechanisms, and access control models, are incapable of providing adequate mechanisms for securing the execution of modern computer systems. Arguably the biggest problem is the large, complex, and omnipotent operating system kernel, which exposes its entire interface to every process in the system. In most cases, the kernel is implemented in an unsafe language and is characterised by a large codebase and rapid evolution. By mounting an attack on one of the operating system components, a local attacker is always just one step away from gaining control over the entire machine.

Operating system level mandatory access control (MAC) security frameworks, like SELinux [108], AppArmor [13], Smack [102], and Solaris Trusted Extensions [41], help to improve the situation and reduce the attack surface. Unfortunately, they have two significant problems. First, these frameworks have to deal with the complex semantics of the OS API, which significantly complicates MAC policies. Second, MAC frameworks cannot prevent privilege escalation attacks that exploit a kernel vulnerability. Neither traditional operating system mechanisms, or MAC frameworks, secure the system against device driver and hardware attacks. Lack of device, and device driver, isolation through the I/O MMU and memory management mechanisms, enable local hardware and software attacks. Capsicum is a recent research effort to implement the capability access control in the FreeBSD kernel at the level of file descriptors [118]. Capsicum is a perfect tool for implementing fine-grained file system access control, and disaggregating privileged user-level applications. However, similar to the MAC frameworks, Capsicum does not protect against a number of attacks on the operating system kernel.

Least privilege microkernel environments The concept of securing a large body of untrusted application code with a small layer of isolation is not new [53]. Multiple projects try to apply this principle in practice in both microkernel [20, 44, 56, 57, 59, 61], and virtual machine [21, 48, 70, 95, 122] based systems. However, there has been no solution that satisfies the needs of both the modern desktop and server environments.

The most promising microkernel project aimed at constructing a secure execution environment is the Trustworthy Systems project at the University of New South West, and NICTA [117]. This project relies on the first formally verified microkernel, seL4 [63], as the basis for isolation and access control. The seL4 user land has many of the same objectives as XCap: capability-based access control, a small layer of isolation, formal reasoning about authority, and disaggregation of the common operating system services. seL4 is an attractive research direction, as it is built upon a formally verified microkernel. However, seL4 targets embedded systems and not the server and desktop market. The verified version of seL4 runs only on the ARM processor and does not provide all features of the industry-grade virtualization platform. Specifically, it is not clear how well the seL4 capability model scales to a large number of objects.

Similar to XCap, both academic and commercial microkernels achieve strong isolation of individual application activities. Many microkernels implement a capability access control interface [52,55,58,63,82,105,106,114], but it is not clear how well these existing implementations scale to the needs of production environments. Academic microkernels do provide virtualization capabilities by running paravirtualized and virtualized systems, but they do not achieve the performance, stability, robustness, and useability of even open-source, industry-grade hypervisors, like Xen [11]. Further, it is not clear if the device driver protection afforded by the I/O MMU mechanisms is sufficiently mature in the microkernel world, as microkernels typically have a smaller commercial community and a smaller footprint in terms of server and cloud installations.

At the moment, none of the microkernel projects offers a set of mature disaggregated system services, secure collaboration mechanisms, and "best practice" recipes for constructing least authority environments. XCap aims to bring theoretical research in the capability access control area to a full-feature, industry-standard hypervisor.

SAFE [79], and Certified OS Kernels [50] are clean slate design projects aimed to build fully verified operating system kernels, and use safer programming languages for the user-level applications. While promising these research efforts do not answer the problem of securing the large stack of existing unmodified applications. In contrast, XCap offers a pragmatic security option for the software stack of server and desktop installation. Further, XCap's disaggregation lessons can be reused later on top of the fully verified platforms.

Virtual machine containers Hypervisors have become a *de-facto* solution for the problem of secure isolation. Multiple commercial projects attempt to secure commodity applications using full-system virtualization (Qubes OS [95], Bromium [21], XenClient [122]). Existing hypervisors succeed in providing complete isolation of applications at the level of hardware virtualization, but they provide no mechanisms to support sharing of resources and collaboration between mutually untrusted applications in a secure way. Complete isolation works well when no sharing is required, e.g. in case of the simple desktop environment. An untrusted desktop application is sealed in its virtual container, and runs until it exits [21,95,122]. However, core operating system services which are inherently designed to provide sharing of resources (e.g. file systems, block storage, network stack), require mechanisms for secure sharing to avoid operating with reduced functionality or with excessive privilege. The latter in particular makes them obvious targets for attacks in server environments and is a likely reason why the above projects do not target security of the server environments.

Disaggregating operating system services, device drivers, and hardware devices is important. Simply launching individual applications inside private virtual machine containers limits the effect of core kernel privilege escalation attacks, but does not address attacks against devices or device drivers. This is because the default configuration of the Xen [11], VMWare [94], and Bromium [21] hypervisors run all device drivers in a single privileged domain. Therefore, a vulnerability in the device driver, or firmware of the device controller, can result in a complete loss of control over the client machine. Similar to XCap, Qubes OS [95] addresses this problem by running device drivers and individual applications in isolated virtual machines. In contrast to XCap, leaving the system services largely monolithic, Qubes OS does not fully minimize their authority. Further, Qubes OS lacks a formal access control model to serve as a foundation for constructing disaggregated applications. Several military-grade certified secure virtualization projects apply the principle of secure isolation to traditional systems [61, 70]. The proprietary nature of these systems limits their impact in the broader community.

4 Details of Proposed Research

4.1 Strong isolation of applications

Challenge #1: Strong isolation of individual VMs The first step of the proposed work is to implement a capability access control interface at the Xen hypervisor level to limit authority of individual virtual machines. We make sure that access to every resource provided by the hypervisor is mediated with a capability check, i.e., the invocation of a hypercall, execution of a privileged instruction, etc. are allowed only if the virtual machine possesses the necessary capability. We implement *isolation by default*—all resources are inaccessable unless a virtual machine was granted a corresponding access capability.

Task 1: Enforcing access control boundary in Xen To simplify analysis of the Xen code, we leverage the work done by the sHype and Xen Security Modules (XSM) projects [29, 96]. XSM is an implementation of the Flask [111] security model for Xen. XSM identifies the complete set of access control objects inside the Xen hypervisor, and all places in the Xen source where security checks involving those objects are required. We reuse this analysis, and implement the capability access control checks as an XSM module.

Task 2: Isolating the I/O path The Xen virtual machine monitor already implements mechanisms for protecting main memory of the system from DMA attacks. We plan to perform a thorough analysis of the Xen I/O MMU capabilities, and work with the Xen open source community to improve the isolation boundary. We plan to extend the Xen I/O MMU interface to work with the capability access control model, and provide a fine-grained isolation of disaggregated device drivers.

4.2 Capability model

Challenge #2: A practical access control model for fine-grained delegation of rights For a majority of realistic applications, the isolation alone unavoidably results in possession of excessive authority, e.g. a file system virtual machine may have access to all files. This immediately makes such an application an attractive attack target. Static isolation has to be extended with a means of managing authority of applications in a dynamic way. The main research challenge for our work is to design a capability model which naturally integrates with a highly-optimized stack of the fully-featured, industry-grade hypervisor, and the system-level stack. Capability-based access control has a long history of research [16–18, 66, 107, 109], and has been implemented in a number of research microkernels [52, 55, 58, 63, 82, 105, 106, 114]. However, it is still unclear how to apply capability access control to a production system in a practical manner.

Task 3: Developing the capability access control model The second step of our work defines a notion of capability, and develops the rules of the capability model. Our main goal is to implement *fast, fine-grained, resource-agnostic* capabilities, accessible from any layer of the system. XCap utilizes a version of the *take-grant* capability access control model, similar to the one used by the seL4 microkernel [37, 38]. The take-grant model was originally suggested by Lipton and Snyder [66], and later developed in many access control systems [16–18, 107, 109]. The take-grant model is *decidable* in linear time [66].

In XCap a capability is just a record in a capability address space of a particular domain. An XCap capability is a mechanism for checking that a certain access is permitted: if a capability record is present in the capability address space of a virtual machine, the access is permitted. Each virtual machine has its own capability address space. We will implement capability spaces inside the Xen hypervisor as sparse tree data structures. The main role of capability address spaces is to provide efficient lookup and insert operations. Each capability has a unique name that is exposed to virtual machines. These unique names enable virtual machines to bind capabilities to high-level objects, e.g. files, network connections, memory pages, and application-defined objects.

Operations on capabilities We base the design of the capability interface on the following pattern of operation: create, grant, and check. In other words, virtual machines can create capabilities, distribute them to other virtual machines, and then use a simple check operation to verify if access is permitted.

Create Any virtual machine, or even any application process inside the virtual machine, can request the Xen hypervisor to create a new capability. Create is implemented as a hypercall, which drops into the Xen hypervisor and creates a new capability object with a unique capability identifier. Initially this capability is inserted in the capability space of the virtual machine which creates it.

Grant The owner of the capability can grant it to any other virtual machine. The capability will be inserted in a capability space of the virtual machine identified by the domain identifier.

Check A virtual machine which implements access control over a resource, can check whether another virtual machine can access a protected resource by invoking the check hypercall. The check hypercall will lookup the specific capability in the capability space of the domain identified by the domain identifier.

Similar to KeyKOS [52], EROS [106], and seL4 [63] we implement a mechanism for the fast hierarchical revocation of capabilities. This way an object granting a capability can always revoke it recursively from all applications.

Formal Reasoning about Authority An important goal of our work is to provide a way to reason about authority of individual applications and security of the system as a whole. To reason about confidentiality and integrity, we want to reuse techniques of existing proofs of confinement in capability systems [66, 107]. These proofs evaluate security properties of a system by computing a transitive closure of the capability graph. To answer confidentiality and integrity questions about a system, the capability layer must provide enough information to describing every data object in the system, and all possible channels, through which information can flow between the objects.

The challenge is to design a system in which its complete state is described with capabilities. While pursuing this general goal, we adhere to several design principles. First, we want our system to support the majority of existing applications with few or no modifications. Second, we want to preserve the performance of the original applications. Third, we want to remain pragmatic and avoid massive re-engineering efforts while implementing the capability layer.

Basic objects and trust We structure the system around a set of *basic objects*, namely those abstractions implemented by the Xen hypervisor: hypercalls, memory pages, event channels, etc. The hypervisor is the only part of the system which must be included in the trusted computing base, a nice property which allows us to reason about confidentiality and integrity properties of the system without the need to trust any code except that of a relatively small hypervisor.

4.3 Building the Least Authority Services

Challenge #3: Practical least authority The third phase of our work is an exercise in building least authority services. We plan to experiment with the capability interface to construct virtual environments in which authority of individual activities is minimized. Our research goal is twofold: first, we would like to test the flexibility of our capability model, second, we aim to come up with a set of primitives designed to simplify construction of the least privileged environments. The practical challenge of our work is to make sure that the majority of a traditional application stack continues running with little or no change.

We use our capability framework to implement a set of small, possibly verifiable components aimed to shield the big, and error-prone codebase of a traditional virtualization stack, components which are critical for implementing collaboration, and therefore end up possessing ambient authority: versioning block storage, network stack, file systems, etc. We use small proxy components to simplify the task of building disaggregated virtual environments in which authority of individual parts is minimal.

Task 4: Disaggregated file system To better understand challenges and design choices behind a process of building a capability system, we look at a file system as an example of a collaboration mechanism with complex semantics, and code organization. To ensure secure isolation of a file system, we run it as a separate virtual machine. Client applications are also isolated inside individual virtual machines and access the file system over the NFS protocol. To explore file system disaggregation, we plan to realize three different file system iterations, each with progressing fine grained isolation properties.

1. The first iteration assumes a design in which the entire file system VM has to be trusted. In such setup, the file system exports file objects, which can be accessed via a capability interface. However, if an attacker compromises the file system virtual machine, he can access and modify the entire file system.

2. The second iteration restructures the file system in a way, in which the file system possesses no authority to access the files, and works only as a naming service for low-level block objects, which are protected by the hypervisor.

3. The third iteration will restructure the file system in a way that clients talk to their own private copy of a file system. Private copies are completely isolated from each other, except for a carefully designed synchronization interface. Such architecture provides strong isolation, eliminates ambient authority from the file system, and potentially leads us to availability guarantees.

Iteration 1: Trusted, monolithic file system We protect a traditional file system with the capability access control model. The following operations need to be implemented.

Binding capabilities to objects The file system client and a part of trusted computing base, which is responsible for performing a security check must agree on the meaning of each capability. To solve this problem, we suggest that a part of the system, which originally creates a particular object binds the capability to its meaning. The file system is a creator of a file object. It is responsible for associating file descriptors with capabilities, and advertise their meaning. The file system can save a capability identifier for each file in a way similar to how a conventional file system stores file access rights. This design is coherent with the implementation of capabilities as file descriptors in the Capsicum project [118].

Access checks In XCap the access to low-level basic objects is always performed by the hypervisor. This provides us with the strong guarantees about confidentiality and integrity of the system. However, our goal is to provide a foundation for constructing the least authority services at any level of the system. To achieve that, XCap allows applications to create their own objects of an arbitrary type. The objects are implemented by an application which creates them, and does not have any special meaning for the hypervisor. An application, which implements a high-level object, relies on support from the capability system to make sure that capabilities are propagated according to the rules of the access control model, remain protected by the hypervisor, unspoofable, and unforgeable. However, the application is responsible for performing the access checks for the objects it manages. In our example, the trusted file system creates capabilities for each file, and performs the access checks on every file access operation.

Presenting capabilities Trying to be pragmatic from the engineering point of view, we aim to reuse the existing Xen IPC mechanisms [46]. In XCap, an application is not required to do anything special to present its capabilities. For the basic objects, the access check is performed by the hypervisor. For high-level objects the check is performed by the application.

Iteration 2: File system as a naming service To further push the principle of least privilege, we redesign a file system to be a naming service for a block layer. Our main goal is to minimize the ambient authority inside the file system. In the first iteration, a trusted file system possesses rights to access the entire disk and all files. This makes the file system a desirable target of a potential security attack.

Now we implement a file system design in which the file system itself possesses no capabilities except the ones required to resolve file names and file descriptors into disk blocks. When a client tries to access a file it grants file system a set of capabilities required to perform a specific operation. Capabilities to access specific disk blocks and modify the file system structure are owned only by the application VMs. When client decides to close a file it revokes its capabilities from the file system. Note that although a file system receives a capability to access a set of disk blocks corresponding to a file, it cannot pass it to another VM, as it does not have a right to grant this capability.

Iteration 3: Pushing least privilege to the extreme The third iteration pushes the principle of least authority to the extreme by redesigning a file system to be a collection of mutually untrusted naming services. Similar to how Asbestos [36] decomposes a web server, we restructure a file system. We create a private copy of a file system in every client virtual machine. This approach ensures a complete isolation of rights between client VMs. In order to ensure consistency of the file system, untrusted servers communicate via a secure synchronization protocol, which is designed to provide guarantees of availability.

More application examples The three-step file system decomposition example provides a general overview of the process of decomposing traditional applications and system services into a set of securely isolated components with the minimal privileges, which are required to perform a specific task. We suggest these three steps for decomposition of all applications in our system. For example, decomposition is of the web server can be achieved by starting a new web-server virtual machine for every incoming request. A more interesting engineering effort is required for decomposition of a web content management system, like wiki. Similar to the trusted file system, a wiki layer is required to implement a set of high-level objects designed to securely manipulate the wiki objects.

4.3.1 Patterns of Secure Collaboration

Task 5: Develop mechanisms of secure collaborarion XCap develops a set of general mechanisms, and techniques aimed at simplifying the general steps of disaggregation.

Proxy domains We plan to experiment with implementing minimal proxy domains aimed to enforce access control on top of existing virtual services. For example, a proxy domain can apply a capability access control to a shared file system. Proxy domains are essentially minimal virtual machines capable of running on top of the Xen virtual machine interface. One possible way to implement proxy domains is to utilize a recent feature of the Xen platform – *Stub Domains*. Stub Domains provide an option to link a traditional POSIX application against a modified version of the libc library, which is capable of running on top of a minimal execution environment provided by the Xen MiniOS [77]. We also plan to experiment with implementing proxy domains in a safe languages. Several language run-times were ported to run as Xen virtual machines [54, 62, 84].

Secure synchronization primitives We plan to develop a set of secure synchronization and communication primitives aimed at providing availability guarantees to the mutually untrusted components. Our work will use techniques of robust composition developed in the capability research community [74, 75, 112].

Tools for managing capabilities The capability access control model provides a powerful mechanism for creating isolated compartments, with strict control of the information flow. We plan to develop a set of principles, libraries, and tools simplifying the task of managing capabilities. Our tools will be specifically targeted at the needs of the desktop and server services, which XCap aims to provide.

4.3.2 Practical Optimizations

Task 6: Fast capabilities and low-overhead VMs To provide a practical execution environment, XCap develops several performance optimizations.

Capability caches We envision running a system with millions of capabilities. It is essential to optimize the capability interface on the critical path, e.g. for the capability check operations. We propose to implement a read-only capability cache inside the memory of the virtual machine, which is responsible for implementing the access checks for the specific type of objects.

Copy-on-write virtual machine memory and file systems To minimize overheads of running large number of virtual machines, XCap relies on the copy-on-write techniques for sharing memory across a large number of almost identical virtual machines. Latest versions of the Xen hypervisor support copy-on-write sharing of virtual machine memory [76]. To eliminate the overheads of disposable root file systems, XCap utilizes a branching storage solution, developed by one of the PIs as part of the time-travel environment for the Emulab network testbed [22].

5 Broader Impacts

Our proposed research will directly benefit the security research and education communities by enhancing software infrastructure that is widely used in those communities. We will release the software developed for this work under an open source license as an extension to the Xen hypervisor. Xen is widely used as a basis for research on virtualization and security [28, 46, 51, 54, 62, 72, 73, 84, 85, 119–121, 124] and has been used to teach classes in those areas. The principle of least authority is widely recognized as a valuable part of the design of secure systems, and our work will serve as a foundation on which to apply this principle to operating systems and applications. As a result, we expect it to enable future research in this area.

The problem of insecure computing environments has large impacts on society as a whole: security breaches lead to violation of privacy, financial fraud, espionage, sabotage, lost productivity, and more. A major reason for the severity of these consequences is that many of our systems simply have too much access to each other: for example, compromise of a word processor can lead to attacks on SCADA systems [15]. By providing strong isolation between applications and services, our proposed work has the potential to mitigate many of the worst problems associated with insecure software.

6 Timeline and Management Plan

PI Regehr will guide the overall direction of the project. co-PI Burtsev will be the development lead. He is is a member of the research staff in the Flux Research Group, an established research group with over 15 years of work on systems and networking. Under PI Regehr's guidance, Burtsev will do much of the day-to-day management and mentoring of the graduate students that we recruit for this work.

Year 1 Our work in the first year will focus on designing and building the basic framework for XCap and building a basic capability-protected service. Specific milestones include: (a) Initial design and implementation of our capability system in Xen, including capability-protected access to memory pages and devices (b) Ability to boot existing unmodified operating systems inside of XCap (c) Basic mechanisms for collaboration between domains (d) Iteration 1 of the filesystem service (trusted filesystem)

Year 2 In the second year, we will examine in more detail the flow of rights in a capability-based virtual systems, build more complex capability-protected services, and begin investigation into formal reasoning about authority. Milestones include: (a) Define a policy language for restricting the flow of capabilities, along the lines of mandatory access controls (MAC) defined by some modern operating systems [13, 41, 102, 108] (b) Initial design and prototype of API proxying through stub domains (c) Iteration 2 of the filesystem (FS as a naming service) (d) Formal reasoning about authority: combined with the policy language, design a framework that, given an initial set of capabilities, reasons about the maximum extent of privilege and information flow. Identify points at which policy could be applied to limit this flow.

Year 3 Our third year will conclude by pushing the principle of least authority to it's extreme: identifying common patterns of secure collaboration between different domains and building systems that divide common services into very small parts with minimal trust. (a) Define a set of common functions for use in construction of secure services in stub domains, enabling the construction of services that do not require a full OS stack to reside in the domain (b) Iteration 3 of the filesystem: collection of mutually-untrusting naming services (c) Identify and implement more services that have are deconstructed into small parts. Possible targets include collaborative authoring environments, safe web browsing with fine-grained isolation, and secure cloud services.

7 Results from Prior NSF Support

Co-PI Burtsev has not previously received NSF support.

Note: *PI Regehr has two current NSF awards related to the Emulab: "CRI: CRD: Keeping Emulab Tuned and Humming" and "MRI: Evolutionary Development of an Advanced Distributed Testbed." These are awards that he took over upon the untimely death of their previous PI, Jay Lepreau. These awards are not related to Dr. Regehr's research and his involvement is purely administrative and at a low level of effort.*

PI Regehr's most closely related prior NSF award is CNS–0615367: "Improving Sensor Network Software Reliability through Language, Tool, and OS Co-Design," in collaboration with Philip Levis and Dawson Engler at Stanford University, September 2006–August 2009. The Utah portion of this grant was $210,000, with $360,000 going to Stanford. Our work was based on the hypothesis that the reliability of TinyOS[1]—an operating system for sensor network nodes—could be significantly improved by adapting the system software, by augmentings its programming language nesC, and by developing static and dynamic checking tools to detect errors.

To detect a common kind of bug that occured at TinyOS component boundaries, we wrote a collection of *contracts* for interfaces and also created a tool for weaving contract checks into application code as it was being compiled. We found and reported a number of bugs and wrote a paper [7] but due to some implementation limitations (clunky integration with the build process, excessive code bloat, and failing to support the newer TinyOS 2) we did not deploy this tool. Our second effort was motivated by the problems associated with type and memory safety bugs on sensor network nodes. The research problem was taking well-known methods for trapping these errors (i.e., type safe versions of C) and making them run on a platform with 4 KB of RAM and very limited I/O. This effort resulted in Safe TinyOS [30] which was deployed in TinyOS starting in version 2.1. The third tool to emerge from our TinyOS work was a model checker and randomized tester called T-Check, which found a number of safety and liveness bugs in TinyOS 2. We released T-Check as open-source software.[2] Fourth, we created Neutron [26] which leveraged type safe execution to create a user/kernel separation in TinyOS applications, permitting applications to reboot without disturbing the kernel, and vice versa.

[1]`http://www.tinyos.net/`
[2]`http://www.cs.utah.edu/~peterlee/tcheck/`

References cited

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity - Principles, Implementations, and Applications. 2005.

[2] Alexander Tereshkin and Rafal Wojtczuk. Introducing Ring -3 Rootkits. 2009.

[3] Alkassar, E. and Paul, W. On the verification of a baby hypervisor for a RISC machine; draft 0, January 2008.

[4] AMD I/O Virtualization Technology (IOMMU). `http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf`.

[5] Angelos Stavrou and Zhaohui Wang. Exploiting Smart-Phone USB Connectivity For Fun And Profit. 2011.

[6] Apple Corporation, Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, National Semiconductor Corporation, Sun Microsystems Inc., and Texas Instruments Inc. 1394 Open Host Controller Interface Specification. 2000.

[7] W. Archer, P. Levis, and J. Regehr. Interface contracts for TinyOS. In *Proc. of the Intl. Conf. on Information Processing in Sensor Networks (IPSN'07), SPOTS Track*, Cambridge, MA, Apr. 2007.

[8] K. Ashcraft and D. R. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[9] Aumaitre D. A little journey inside windows memory. *Journal in computer virology*, 5(2):105–117, 2009.

[10] Aumaitre D. and Devine C. Subverting windows 7 x64 kernel with dma attacks. *HITBSecConf 2010 Amsterdam*, 29, 2010.

[11] Barham, P. and Dragovic, B. and Fraser, K. and Hand, S. and Harris, T. and Ho, A. and Neugebauer, R. and Pratt, I. and Warfield, A. Xen and the art of virtualization. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 164–177. ACM, 2003.

[12] D. Barrall and D. Dewey. "Plug and Root," the USB Key to the Kingdom. 2005.

[13] Bauer, M. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, 2006(148):13, 2006.

[14] Becher, M., Dornseif, M., and Klein, C.N. Firewire all your memory are belong to us. *Proceedings of CanSecWest*, 2005.

[15] Bencsáth, B. and Pék, G. and Buttyán, L. and Félegyházi, M. Duqu: Analysis, Detection, and Lessons Learned. In *2012 European Workshop on System Security (EuroSec)*, 2012.

[16] M. Bishop. Hierarchical take-grant protection systems. In *ACM SIGOPS Operating Systems Review*, volume 15, pages 109–122. ACM, 1981.

[17] M. Bishop. Conspiracy and information flow in the take-grant protection model. *Journal of Computer Security*, 4(4):331–359, 1996.

[18] Bishop, M. and Snyder, L. The transfer of information and authority in a protection system. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 45–54. ACM, 1979.

[19] Boileau A. Hit by a bus: Physical access attacks with firewire. *Presentation, Ruxcon*, 2006.

[20] Bomberger, A.C. and Frantz, A.P. and Frantz, W.S. and Hardy, A.C. and Hardy, N. and Landau, C.R. and Shapiro, J.S. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, 1992.

[21] Bromium. Bromium Micro-virtualization. *whitepaper*, 2010. `http://www.bromium.com/misc/BromiumMicrovirtualization.pdf`.

[22] A. Burtsev, P. Radhakrishnan, M. Hibler, and J. Lepreau. Transparent checkpoints of closed distributed systems in Emulab. In *Proc. of the 4th ACM European Conference on Computer Systems*, pages 173–186, Nuremberg, Germany, Apr. 2009.

[23] L. Butti and J. Tinns. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology*, 4:25–37, 2008.

[24] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM. `http://doi.acm.org/10.1145/2103799.2103805`.

[25] K. Chen. Reversing and exploiting an Apple firmware update. 2009. `http://www.blackhat.com/presentations/bh-usa-09/CHEN/BHUSA09-Chen-RevAppleFirm-PAPER.pdf`.

[26] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving sensor network software faults. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, Big Sky, MT, USA, Oct. 2009.

[27] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An Empirical Study of Operating System Errors. *Operating Systems Review*, 35:73–88, 2001.

[28] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews. Xen and the art of repeated research. In *Proceedings of the USENIX Annual Technical Conference*, Nov. 2004.

[29] G. Coker. Xen security modules (xsm). *Xen Summit*, pages 1–33, 2006. `http://mail.xen.org/files/summit_3/coker-xsm-summit-090706.pdf`.

[30] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proc. of the 5th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 205–218, Sydney, Australia, Nov. 2007.

[31] Coverity, Inc. Coverity SAVE, 2012. `http://www.coverity.com/products/coverity-save.html`.

[32] C. Cowan, C. Pu, D. Maier, H. Hinton, and J. Walpole. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[33] M. Dahlin, R. Johnson, R. Krug, M. McCoyd, S. Ray, and B. Young. Toward the verification of a simple hypervisor. In *10th International Workshop on the ACL2 Theorem Prover and its Applications*, Nov. 2011.

[34] Dennis, J.B. and Van Horn, E.C. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[35] Duflot L., Perez Y.A., Valadon G., and Levillain O. Can you still trust your network card. *CanSecWest/core10*, pages 24–26, 2010.

[36] Efstathopoulos, P. and Krohn, M. and VanDeBogart, S. and Frey, C. and Ziegler, D. and Kohler, E. and Mazieres, D. and Kaashoek, F. and Morris, R. Labels and event processes in the Asbestos operating system. *ACM SIGOPS Operating Systems Review*, 39(5):17–30, 2005.

[37] Elkaduwe, D. *A principled approach to kernel memory management.* PhD thesis, University of New South Wales, 2010.

[38] Elkaduwe, D., Klein, G., and Elphinstone, K. Verified protection model of the seL4 microkernel. *Verified Software: Theories, Tools, Experiments*, pages 99–114, 2008.

[39] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 1–16, San Diego, CA, Oct. 2000. USENIX Association.

[40] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Operating Systems Design and Implementation*, pages 75–88, 2006.

[41] Faden, G. Solaris trusted extensions. *Sun Microsystems Whitepaper, April*, 2006.

[42] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. 2011. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`.

[43] Fernand Lone Sang, Vincent Nicomette, and Yves Deswarte. Demonstration of a peer-to-peer DMA Attack against the Framebuffer of a Graphic Controller Through FireWire. 2011.

[44] Feske, N. and Helmuth, C. *Design of the Bastei OS architecture.* Techn. Univ., Fakultät Informatik, 2007.

[45] sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks. `http://www.crysys.hu/skywiper/skywiper.pdf`.

[46] Fraser, K. and Hand, S. and Neugebauer, R. and Pratt, I. and Warfield, A. and Williamson, M. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[47] A. Ganapathi, V. Ganapathi, and D. A. Patterson. Windows XP Kernel Crash Analysis. In *USENIX Systems Administration Conference*, pages 149–159, 2006.

[48] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles*, pages 193–206, 2003.

[49] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. C. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *ACM Symposium on Operating Systems Principles*, pages 103–116, 2009.

[50] Gu, L., Vaynberg, A., Ford, B., Shao, Z., and Costanzo, D. CertiKOS: a certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 3. ACM, 2011.

[51] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine:harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation*, Dec. 2008.

[52] Hardy, N. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[53] Härtig, H. Security architectures revisited. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 16–23. ACM, 2002.

[54] Haskell Lightweight Virtual Machine (HaLVM). `http://corp.galois.com/halvm`.

[55] Heiser, G. and Leslie, B. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010.

[56] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[57] Herder, J.N. and Bos, H. and Gras, B. and Homburg, P. and Tanenbaum, A.S. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[58] Hohmuth, M. and Tews, H. The VFiasco approach for a verified operating system. In *Proc. 2nd ECOOP Workshop on Programm Languages and Operating Systems*. Citeseer, 2005.

[59] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22. ACM, 2004.

[60] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*, pages 298–307, 2004.

[61] INTEGRITY Real-Time Operating System. http://www.ghs.com/products/rtos/integrity.html.

[62] Project Kenai. http://kenai.com/.

[63] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., and others. seL4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[64] J. Larimer. Beyound Autorun: Exploiting vulnerabilities with removable storage. 2011. https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabiliters%20w-removeable%20storage-Slides.pdf.

[65] J. Larimer. USB Autorun attacks against Linux. 2011. http://blogs.iss.net/archive/papers/ShmooCon2011-USB_Autorun_attacks_against_Linux.pdf.

[66] Lipton, R.J. and Snyder, L. A linear time algorithm for deciding subject security. *Journal of the ACM (JACM)*, 24(3):455–464, 1977.

[67] LNK-vulnerability-CVE-2010-2568, 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2568.

[68] Loc Duflot and Yves-Alexis Perez. Can you still trust your network card? In *CanSecWest*, 2010.

[69] LSI Corporation, Hewlet-Packard Company, Intel Corporation, Microsoft Corporation, NEC Corporation, and ST-NXP Wireless Company. On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification. 2009.

[70] Desktop Virtualization and Secure Client Virtualization Based on Military-Grade Technology. http://www.lynuxworks.com/virtualization/secure-client-virtualization.php.

[71] D. M. 0wned by an iPod. *Presentation, PacSec*, 2004. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6092759.

[72] McDermott, J. and Freitas, L. A formal security policy for xenon. In *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, pages 43–52. ACM, 2008.

[73] McDermott, J. and Kirby, J. and Montrose, B. and Johnson, T. and Kang, M. Re-engineering Xen internals for higher-assurance security. *Information Security Technical Report*, 13(1):17–24, 2008.

[74] Miller, M. and Tribble, E. and Shapiro, J. Concurrency among strangers. *Trustworthy Global Computing*, pages 195–229, 2005.

[75] Miller, M.S. and Shapiro, J.S. *Robust composition: Towards a unified approach to access control and concurrency control.* PhD thesis, Johns Hopkins University, 2006.

[76] Miłós, G. and Murray, D.G. and Hand, S. and Fetterman, M.A. Satori: Enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 1–1. USENIX Association, 2009.

[77] Mini-OS. `http://wiki.xen.org/wiki/Mini-OS`.

[78] Moritz Jodeit and Martin Johns. USB Device Drivers: A Stepping Stone into Your Kernel. In *European Conference on Computer Network Defense*, 2010.

[79] Morrisett, J.G. and DeHon, A. and Karel, B. and Malecha, G.M. and Montagu, B. and Morisset, R. and Pierce, B.C. and Pollack, R. and Ray, S. and Shivers, O. and others. Preliminary Design of the SAFE Platform. In *Proceedings of the 6th workshop on programming languages and operating systems. New York: Association for Computing Machinery*, 2011.

[80] T. Mueller. Virtualised USB Fuzzing for Vulnerabilities. 2010. `https://muelli.cryptobitch.de/paper/2010-usb-fuzzing.pdf`.

[81] MWR InfoSecurity. Linux USB Device Driver Buffer Overflow. 2009. `http://labs.mwrinfosecurity.com/assets/173/mwri_linux-usb-buffer-overflow_2009-10-29.pdf`.

[82] Neumann, P., Boyer, R.S., Feiertag, R.J., Levitt, K.N., and Robinson, L. *A provably secure operating system: The system, its applications, and proofs.* SRI International, 1980.

[83] Niels Provos and Panayiotis Mavrommatis and Moheeb Abu Rajab and Fabian Monrose. All Your iFRAMEs Point to Us. In *USENIX Security Symposium*, pages 1–16, 2008.

[84] Mirage: extreme specialisation of virtual appliances. `http://www.xen.org/xensummit/xs12na_talks/M10b.html`.

[85] P. Padala and X. Zhu. Automated control of multiple virtualized resources. In *Proceedings of the EuroSys 2009 Conference*, 2009.

[86] N. Palix, G. Thomas, S. Saha, C. Calvs, J. L. Lawall, and G. Muller. Faults in linux: ten years later. In *Architectural Support for Programming Languages and Operating Systems*, pages 305–318, 2011.

[87] The Rise of PDF Malware. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_rise_of_pdf_malware.pdf`.

[88] Bypassing StackGuard and StackShield. Phrack Magazine. Volume 0xa. Issue 0x38.

[89] PRE-SA-2011-01. Multiple Linux kernel vulnerabilities in partition handling code of LDM and MAC partition tables., 2011.

[90] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., and Modadugu, N. and others. The ghost in the browser analysis of web-based malware. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, 2007.

[91] A Tale of Two Pwnies (Part 1). `http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html`.

[92] A Tale Of Two Pwnies (Part 2). `http://blog.chromium.org/2012/06/tale-of-two-pwnies-part-2.html`.

[93] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012. `http://doi.acm.org/10.1145/2133375.2133377`.

[94] Rosenblum, M. VMwares Virtual Platform. In *Proceedings of Hot Chips*, pages 185–196, 1999.

[95] Rutkowska, J. and Wojtczuk, R. Qubes OS architecture. *Invisible Things Lab Tech Rep*, 2010.

[96] Sailer, R., Valdez, E., Jaeger, T., Perez, R., Van Doorn, L., Griffin, J.L., and Berger, S. sHype: Secure hypervisor approach to trusted virtualized systems. *Techn. Rep. RC23511*, 2005.

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

[98] Sang F.L., Lacombe E., Nicomette, V., and Deswarte, Y. Exploiting an IO MMU vulnerability. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 7–14. IEEE, 2010.

[99] Sang, F.L., Lacombe, E., Nicomette, V., and Deswarte, Y. Exploiting an IO MMU vulnerability. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 7 –14, oct. 2010.

[100] Sang, F.L., Nicomette, V., and Deswarte, Y. I/O Attacks in Intel PC-based Architectures and Countermeasures. In *SysSec Workshop (SysSec), 2011 First*, pages 19 –26, july 2011.

[101] Sang, F.L., Nicomette, V., Deswarte, Y., and Duflot, L. Attaques DMA peer-to-peer et contremesures. In *Proceedings of the Symposium sur la Sécurité des Technologies de LInformation et des Communications (SSTIC)*, 2011.

[102] Schaufler, C. Smack in embedded computing. In *Proceedings of the 10th Linux Symposium*, 2008.

[103] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM. `http://doi.acm.org/10.1145/1294261.1294294`.

[104] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*, pages 552–561, 2007.

[105] Shapiro, J.S., Northup, E. and Doerrie, M.S., Sridhar, S., Walfield, N.H., and Brinkmann, M. Coyotos microkernel specification. *The EROS Group, LLC, 0.5 edition*, 2007.

[106] Shapiro, J.S., Smith, J.M., and Farber, D.J. *EROS: a fast capability system*, volume 33. ACM, 1999.

[107] Shapiro, J.S. and Weber, S. Verifying the EROS confinement mechanism. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 166–176. IEEE, 2000.

[108] Smalley, S. and Vance, C. and Salamon, W. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1:43, 2001.

[109] Snyder, L. Theft and conspiracy in the Take-Grant protection model. 1980.

[110] SOLAR DESIGNER. Stackpatch, 1998. `http://www.openwall.com/`.

[111] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proc. of the 8th USENIX Security Symp.*, pages 123–139, Aug. 1999.

[112] Spiessens, F. and Van Roy, P. The Oz-E project: Design guidelines for a secure multiparadigm programming language. *Multiparadigm Programming in Mozart/Oz*, pages 21–40, 2005.

[113] Remotely Attacking Network Cards (or why we do need VT-d and TXT). `http://theinvisiblethings.blogspot.com/2010/04/remotely-attacking-network-cards-or-why.html`.

[114] Steinberg, U. and Kauer, B. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010.

[115] Tews, H. and Jacobs, B. and Poll, E. and van Eekelen, M. and van Rossum, P. Specification and verification of the Nova microhypervisor. Technical report, Citeseer, 2007.

[116] A. Triulzi. The Jedi Packet Trick takes over the Deathstar: taking NIC backdoors to the next level. In *CanSecWest*, 2010.

[117] Trustworthy Systems project. `http://ssrg.nicta.com.au/projects/TS/about.pml`.

[118] Watson, R.N.M. and Anderson, J. and Laurie, B. and Kennaway, K. Capsicum: practical capabilities for UNIX. In *USENIX Security*, 2010.

[119] T. Wood, A. Lagar-Cavilla, K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. PipeCloud: Using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *Proceedings of 2nd Symposium on Cloud Computing (SOCC)*, Oct. 2011.

[120] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. CloudNet: Dynamic pooling of cloud resources by live wan migration of virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, Mar. 2011.

[121] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007.

[122] XenClient. `http://www.citrix.com/products/xenclient/how-it-works.html`.

[123] Yang, J. and Hawblitzel, C. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, 2010.

[124] F. Zhang, J. Chen, and B. Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[125] Zongwei Zhou and Virgil D. Gligor and James Newsome and Jonathan M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *IEEE Symposium on Security and Privacy*, pages 616–630, 2012.