Appears in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010

Trust and Protection in the Illinois Browser Operating System

Shuo Tang, Haohui Mai, Samuel T. King
University of Illinois at Urbana-Champaign

Abstract

Current web browsers are complex, have enormous trusted computing bases, and provide attackers with easy access to modern computer systems. In this paper we introduce the Illinois Browser Operating System (IBOS), a new operating system and a new browser that reduces the trusted computing base for web browsers. In our architecture we expose browser-level abstractions at the lowest software layer, enabling us to remove almost all traditional OS components and services from our trusted computing base by mapping browser abstractions to hardware abstractions directly. We show that this architecture is flexible enough to enable new browser security policies, can still support traditional applications, and adds little overhead to the overall browsing experience.

1 Introduction

Web-based applications (web apps), browsers, and operating systems have become popular targets for attackers of computer systems. Vulnerabilities in web apps are widespread and increasing. For example, cross-site scripting (XSS), which is effectively a form of script injection into a web app, recently overtook the ubiquitous buffer overflow as the most common security vulnerability [50]. Vulnerabilities in web browsers are less common than web app vulnerabilities, but still occur often. For example, in 2009 Internet Explorer, Chrome, Safari, and Firefox had 349 new security vulnerabilities [4], and attackers exploit browsers commonly [53, 37, 42, 41, 4]. Vulnerabilities in libraries, system services, and operating systems are less common than vulnerabilities in browsers, but are still problematic for modern systems. For example, glibc, GTK+, X, and Linux had 114 new security vulnerabilities in 2009 [1], and in 2009 the most commonly attacked vulnerability was a remote code execution bug in the Windows kernel [4].

However, not all attacks on web apps, browsers, and operating systems are equally virulent. At the top of the computer stack, attacks on web apps, such as XSS, operate within current browser security policies that contain the damage to the vulnerable web app. Moving down the computer stack, attacks on browsers can cause more damage because a successful attack gives the attacker access to browser data for all web apps and access to other resources on the system. At the lowest layers of the computer stack, attacks on libraries, shared system services, and operating systems are the most serious attacks because attackers can access arbitrary states and events, giving them complete control of the system.

Overall, these trends indicate that vulnerabilities higher in the computer stack are more common, but vulnerabilities lower in the computer stack provide attackers with more control and are more damaging. In this paper we focus on preventing and containing attacks on browsers, libraries, system services, and operating systems – the lower layers of the computer stack.

Current research efforts into more secure web browsers help improve the security of browsers, but remain susceptible to attacks on lower layers of the computer stack. The OP web browser [26], Gazelle [52], Chrome [11], and ChromeOS [25] propose new browser architectures for separating the functionality of the browser from security mechanisms and policies. However, these more secure web browsers are all built on top of commodity operating systems and include complex user-mode libraries and shared system services within their trusted computing base (TCB). Even kernel designs with strong isolation between OS components (e.g., microkernels [24, 27, 28] and information-flow kernels [18, 57, 33]) still have OS services that are shared by all applications, which attackers can compromise and still cause damage. Here are a few ways that an attacker can still cause damage to more secure web browsers built on top of traditional OSes:

- A compromised Ethernet driver can send sensitive HTTP data (e.g., passwords or login cookies) to any remote host or change the HTTP response data before routing it to the network stack.

- A compromised storage module can modify or steal any browser related persistent data.

- A compromised network stack can tamper with any network connection or send sensitive HTTP data to an attacker.

- A compromised window manager can draw any content on top of a web page to deploy visual attacks, such as phishing.

In this paper we describe IBOS, an operating system and a browser co-designed to reduce drastically the TCB for web browsers and to simplify browser-based systems. Our key insight is that our lowest-layer software can expose browser-level abstractions, rather than general-purpose OS abstractions, to provide vastly improved security properties for the browser *without* affecting the TCB for traditional applications. Some examples of browser abstractions are cookies for persistent storage, hypertext transfer protocol (HTTP) connections for network I/O, and tabs for displaying web pages. To support traditional applications, we build UNIX-like abstractions on top of our browser abstractions.

IBOS improves on past approaches by removing typically shared OS components and system services from our browser's TCB, including device drivers, network protocol implementations, the storage stack, and window management software. All of these components run above a trusted *reference monitor* [9], which enforces our security policies. These components operate on browser-level abstractions, allowing us to map browser security policies down to the lowest-level hardware directly and to remove drivers and system services from our TCB.

This architecture is a stark contrast to current systems where *all* applications layer application-specific abstractions on top of general-purpose OS abstractions, inheriting the cruft needed to implement and access these general OS abstractions. By exposing application-specific abstractions at the OS layer, we can cut through complex software layers for one particular application without affecting traditional applications adversely, which still run on top of general OS abstractions and still inherit cruft. We choose to illustrate this principle using a web browser because browsers are used widely and have been prone to security failures recently. Our goal is to build a system where a user can visit a trusted web site safely, even one or more of the components on the system have been compromised.

Our contributions are:

- IBOS is the first system to improve browser and OS security by making browser-level abstractions first-class OS abstractions, providing a clean separation between browser functionality and browser security.

- We show that having low-layer software expose browser abstractions enables us to remove almost all traditional OS components from our TCB, including device drivers and shared OS services, allowing IBOS to withstand a wide range of attacks.

- We demonstrate that IBOS can still support traditional applications that interact with the browser and shared OS services without compromising the security of our system.

2 The IBOS architecture

This paper presents the design and implementation of the IBOS operating system and browser that reduce the TCB for browsing drastically. Our primary goals are to enforce today's browser security policies with a small TCB, without restricting functionality, and without slowing down performance. To withstand attacks, IBOS must ensure any compromised component (1) cannot tamper with data it should not have access to, (2) cannot leak sensitive information to third parties, and (3) cannot access components operating on behalf of different web sites.

In this section we discuss the design principles that guide our design and the overall system architecture. In Section 4 we discuss the security policies and mechanisms we use.

2.1 Design principles

We embrace microkernel [27], Exokernel [19], and safety kernel design principles in our overall architecture. By combining these principles with our insight about exposing browser abstractions at the lowest software layer we hope to converge on a more trustworthy browser design. Five key principles guide our design:

1. *Make security decisions at the lowest layer of software.* By pushing our security decisions to the lowest layers we hope to avoid including the millions of lines of library and OS code in our TCB.

2. *Use controlled sharing between web apps and traditional apps.* Sharing data between web apps and traditional apps is a fundamental functionality of today's practical systems and should be supported. However, this sharing should be facilitated through a narrow interface to prevent misuse.

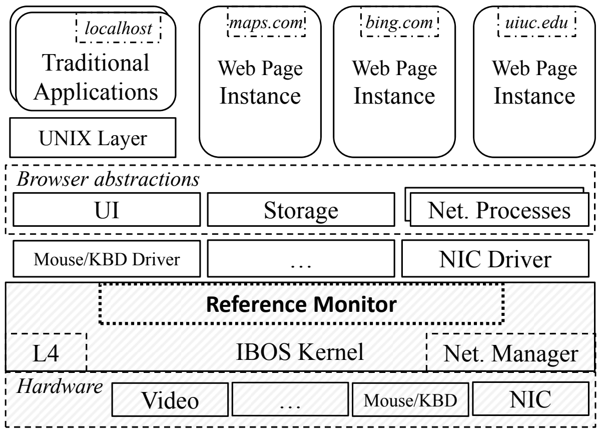

Figure 1: Overall IBOS architecture. Our system contains user-mode drivers, browsers API managers, web page instances, and traditional processes. To manage the interactions between these components, we use a reference monitor that runs within our IBOS kernel. Shaded regions make up the TCB.

3. *Maintain compatibility with current browser security policies.* Our primary goal is to improve the enforcement of current browser policies without changing current web-based applications.

4. *Expose enough browser states and events to enable new browser security policies.* In addition to enforcing current browser policies, we would like our architecture to adapt easily to future browser policies.

5. *Avoid rule-based OS sandboxing for browser components.* Fundamentally, rule-based OS sandboxing is about restricting unused or overly permissive interfaces exposed by today's operating systems. However, sandboxing systems can be complex (the Ubuntu 10.04 SELinux reference policy uses over 104K lines of policy code) and difficult to implement correctly [23, 51]. If our architecture requires OS sandboxing for browser components then we should rethink the architecture.

2.2 Overall architecture

Figure 1 shows the overall IBOS architecture. The IBOS architecture uses a basic microkernel approach with a thin kernel for managing hardware and facilitating message passing between processes. The system includes user-mode device drivers for interacting directly with hardware devices, such as network interface cards (NIC), and browser API managers for accessing the drivers and implementing browser abstractions. The key browser abstractions that the browser API managers implement are HTTP requests, cookies and local storage for storing persistent data, and tabs for displaying user-interface (UI) content. Web apps use these abstractions directly to implement browser functionality, and traditional applications (traditional apps) use a UNIX layer to access UNIX-like abstractions on top of these browser abstractions.

2.2.1 The IBOS kernel

Our IBOS kernel is the software TCB for the browser and includes resource management functionality and a reference monitor for security enforcement. The IBOS kernel also handles many traditional OS tasks such as managing global resources, creating new processes, and managing memory for applications. To facilitate message passing, the IBOS kernel includes the L4Ka::Pistachio [8] message passing implementation and MMU management functions. All messages pass through our reference monitor and are subjected to our overall system security policy. Section 4 describes the policies that the IBOS kernel enforces and the mechanisms it uses to implement these policies.

2.2.2 Network, storage, and UI managers

The IBOS network subsystem handles HTTP requests and socket calls for applications. To handle HTTP requests, *network processes* check a local cache to see if the request can be serviced via the cache, fetch any cookies needed for the request, format the HTTP data into a TCP stream, and transform that TCP stream into a series of Ethernet frames that are sent to the NIC driver. *Socket network processes* export a basic socket API and simply transform TCP streams to Ethernet frames for transmission across the network. Only traditional apps can access our socket network processes. The IBOS kernel manages global states, like port allocation.

The IBOS storage manager maintains persistent storage for key-value data pairs. The browser uses the storage manager to store HTTP cookies and HTML5 local storage objects, and the basic object store includes optional parameters, such as `Path` and `Max-Age`, to expose cookie properties to the reference monitor. The storage manager uses several different namespaces to isolate objects from each other. Web apps and network processes share a namespace based on the origin (the `<protocol, domain name, port>` tuple of a uniform resource locator) that they originate from, and web apps and traditional apps share a "localhost" namespace, which is separate from the HTTP namespace. All other drivers and managers have their own private namespaces to access persistent data.

The IBOS UI manager plays the role of the window manager for the system. However, rather than implement the browser UI components on top of the traditional window motif, we opted for a tabbed browser motif. Basic browser UI widgets, called the browser chrome, are displayed at the top of the screen. IBOS displays web pages in tabs and the user can have any number of tabs open for web apps. There is a tab for basic browser configuration and administration, and a tab that is shared by traditional apps. If traditional apps wish to implement the window motif, they can do so within the tab. The main advantage of our browser-based motif is that it enables IBOS to bypass the extra layers of indirection traditional window managers put between applications and the underlying graphics hardware, exposing browser UI elements and events directly to the IBOS kernel. We discuss the security implications of our design decision in more detail in Section 4.8.

2.2.3 Web apps, traditional apps, and plugins

The IBOS system supports two different types of processes: web page instances and traditional processes. A web page instance is a process that is created for each individual web page a user visits. Each time the user clicks on a link or types a uniform resource locator (URL) into the address bar, the IBOS kernel creates a new web page instance. Web page instances are responsible for issuing HTTP requests, parsing HTML, executing JavaScript, and rendering web content to a tab. Traditional processes can execute arbitrary instructions, and the key difference between a web page instance and a traditional processes is that the IBOS kernel gives them different security labels, which the kernel uses for access control decisions. Web page instances are labeled with the origin of the HTTP request used to initiate the new web page, and traditional processes are labeled as being from "localhost." These two processes interact via the storage subsystem since both types of processes can access "localhost" data.

In general, plugins are external applications that browsers use to render non-HTML content. One common example of a plugin is the Flash player that enables browsers to play Flash content. In IBOS, plugins run as traditional processes, except that they are launched by the browser and the system gives them access to browser states and events through a standard plugin programming interface, called the NPAPI [2].

3 Current browser policies

In this section we give a brief introduction to the same-origin policy (SOP) for browser security. For a more complete discussion of this policy and others, plus experimental results showing how current browsers implement them, please see a recent paper by Singh, *et al.* [47].

The primary security policy that all modern browsers implement is the SOP. The SOP acts as a non-interference policy for the web. Loosely speaking, the SOP provides isolation for web pages and states that come from different origins – origins are used as labels for browser access control policies. If the browser has a web page open from `uiuc.edu` and from `attacker.com`, the SOP should ensure that these two web pages are isolated from each other. Unfortunately, Chrome, IE8, Safari, and Firefox all enforce the SOP using a number of checks scattered throughout the millions of lines of browser code and current browsers have had trouble implementing the SOP correctly [14].

In a browser, a frame is a container that encapsulates a HTML document and any material included in that HTML document. Web pages are frames, and web developers can embed additional frames within web pages – these frames are called `iframes.` Developers can include `iframes` from the same origin as the hosting frame, or from a different origin. Each frame is labeled with the origin of the main HTML document used to populate the frame, meaning that a cross-origin `iframe` has a different label than the hosting web page.

In general HTML documents include references to network objects that the browser will download and display to form the web page. These network objects can be images, JavaScript, and CSS. Browsers can download these objects from any domain and the browser labels them with the origin of the hosting frame. For example, if a page from `uiuc.edu` includes a script from `foo.com`, that script runs with full `uiuc.edu` permissions and can access any of the states in that web page. Browsers can also download HTML documents and XML HTTP requests (used for Ajax), but the SOP dictates that these objects must come from the same origin as the hosting frame.

4 IBOS security policies and mechanisms

Our primary goal is to enforce browser security policies from within our IBOS kernel. This section describes the mechanisms that the IBOS kernel uses to enforce the SOP. We also discuss policies and mechanisms for enforcing UI interactions, and we describe a custom policy engine that lets web sites further restrict current policies.

4.1 Threat model and assumptions

Our primary goal is to ensure that the IBOS kernel upholds our security policies even if one or more of the subsystems have been compromised. In our threat model,

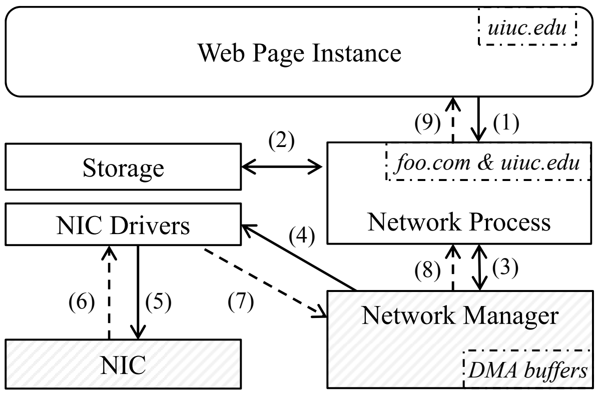

Figure 2: This figure enlarges the right half of Figure 1 and shows how our IBOS subsystems interact when a web page instance from `uiuc.edu` issues a network request to `foo.com`. Subsystems are shown in boxes and solid and dotted arrows represent IBOS messages for outgoing and incoming data respectively. The reference monitor (which is not shown here) checks all these messages to enforce security properties.

we assume that an attacker controls a web site and can serve arbitrary data to our browser, or that the system contains a malicious traditional app. We also assume that this malicious data or traditional app can compromise one or more of the components in our system. These susceptible components include all drivers, browser API managers, web page instances, and traditional processes. Once the attacker takes control of these components, we assume that he or she can execute arbitrary instructions as a result of the attack. We focus on maintaining the integrity and confidentiality of the data in our browser. In other words, we would like the user to be able to open a web page on a trusted web server, and interact with this web page securely, even if everything on the client system outside of our TCB has been compromised. Availability is an important, but separate, aspect of browser security that we do not address in this paper.

In our system we trust the layers upon which we built IBOS. These layers include the IBOS kernel and the underlying hardware. Like all other browsers, IBOS predicates security decisions based on domain names, so we trust domain name servers to map domain names to IP addresses correctly. Compromising any of these trusted layers compromises the security of IBOS.

4.2 IBOS work flow

This section describes a web page instance making a network request to help illustrate the security mechanisms that IBOS uses.

Figure 2 shows the flow of how a web page instance fetches data from the network. The user visits a page hosted at `uiuc.edu` and this web page includes an image from `foo.com`. To download the image, (1) the web page instance will make an HTTP request that the IBOS kernel forwards to an appropriate network process. The network process forms a HTTP request, which includes setting up HTTP headers, (2) fetching cookies from the storage subsystem, (3) requesting a free local TCP port to transform this request into TCP/IP packets and Ethernet frames, and (4) sending it to network manager. The network manager notifies the Ethernet driver which (5) programs the NIC to transmits the packet out to the network. When the NIC receives a reply for the request, (6) it notifies the Ethernet driver. The driver subsequently (7) notifies the network manager, which (8) forwards the packet to the appropriate network process. The network process then parses the data and (9) passes the resulting HTTP reply and data to the original web page instance.

4.3 IBOS labels

To enforce access control decisions, the IBOS kernel labels web page instances, traditional processes, and network processes. IBOS labels specify the resources that a process can access or messages it can receive. Each web page instance has one label, which is the origin of the main HTML document. Each traditional process is labeled as being from "localhost" when they are created. Each network process has an origin label for the network resources it handles and has an origin label for the web page instances that are allowed to access it. IBOS labels the processes upon creation, and keeps the labels unchanged throughout the processes' life-cycle.

An important point is that the IBOS kernel infers the origin labels for web page instances and network processes automatically by extracting related information from the messages passed among them. By inferring labels rather than relying on processes to label themselves, the IBOS kernel ensures that it has the correct label information, even if a process is compromised.

The *newUrl* and *fetchUrl* IBOS system calls are the two requests that cause the kernel to label processes. The *newUrl* system call is used by web page instances and the UI manager use to navigate the browser to a new URL. The *newUrl* system call consists of two arguments: a URL and a byte array for HTTP POST data. When the IBOS kernel receives a *newUrl* request it will create a new web page instance and set the label for this web page instance by parsing the origin out of the URL argument of the *newUrl* request. When servicing *newUrl* requests, the IBOS kernel will reuse old web page instances (to reduce process startup times), but only when the origin labels match for the old web page instance and the URL argument.

Web page instances use the *fetchUrl* system call to issue HTTP and HTTPS requests to fetch network objects, such as images. The *fetchUrl* system call has two arguments: a URL and HTTP header information. When a web page instance issues a *fetchUrl* system call, the IBOS kernel uses the origin of the web page instance (set by the original *newUrl* call) and the origin of the *fetchUrl* URL argument to find a network process with these same labels, or creates a new network processes and labels it accordingly if an existing network process cannot be found.

More details about how we use these labels for access control decisions are described in the remainder of this section.

4.4 Security invariants

For all of our subsystems, we use *security invariants* that are assertions on all interactions between subsystems that check basic security properties. The key to our security invariants is that we can extract security relevant information from messages automatically, and provide high assurance that the system maintains the security policy without having to understand how each individual subsystem is implemented. Using these security invariants, we remove from the TCB almost all of the components found in modern commodity operating systems, including device drivers.

The ideal security invariant is complete, implementation agnostic, executes quickly, and requires only a small amount of code in the IBOS kernel. A complete invariant can infer all of the states needed to ensure the high-level security policy, and an implementation agnostic invariant can infer states without relying on the specific implementation of individual subsystems. The IBOS kernel evaluates invariants in the kernel and inline with messages, so security invariants should execute quickly and require little code to implement. In our design we strive to make the appropriate trade offs among these properties to improve security without making the system slow or increasing our TCB significantly. The base security invariant we have is:

SI 0: *All components can only perform their designated functions.*

For example, the UI subsystem can never ask for cookie data or the storage manager cannot impersonate a network process to send synthesized attack HTTP data to a web page instance.

4.5 Driver invariants

The two driver invariants the IBOS kernel enforces are:

SI 1: *Drivers cannot access DMA buffers directly.*
SI 2: *Devices can only access validated DMA buffers.*

In our approach, we use a split driver architecture where we separate the management of device control registers from the use of device buffers (SI 1). For example, our Ethernet driver never has access to transmit or receive buffers directly. Instead, it knows the physical addresses where the IBOS kernel stores these buffers, and it programs the NIC to use them. By separating these two functions we can interpose on the communications between them to ensure that IBOS upholds browser security policies, even if an attacker completely compromises a shared driver.

Using this split architecture, processes fill in device-specific buffers for DMA transfers, and the IBOS kernel infers when drivers initiate DMA transfers to ensure that the driver instructs the device to use a verified DMA buffer (SI 2). Fortunately, DMA buffers tend to use well-defined interfaces, like Ethernet frames for Ethernet drivers, so the IBOS kernel can readily glean security relevant information from these DMA buffers before the device accesses them. Unfortunately, the interface between drivers and devices is device-specific, so the IBOS kernel must have a small state machine for each device to properly infer DMA transfers. However, we found this state machine to be quite small for the devices that we use in IBOS.

In IBOS we implement a driver for the e1000 NIC, a VESA BIOS Extensions driver for our video card, and drivers for the mouse and keyboard.

4.6 Storage invariants

The primary invariant we strive to enforce in the storage manager is:

SI 3: *All of our key-value pairs maintain confidentiality and integrity even if the storage stack itself becomes compromised.*

To enforce this invariant, our IBOS kernel encrypts all objects before passing them to the storage subsystem. To encrypt data, the IBOS kernel maintains separate encryption keys for all of the namespaces on the IBOS system. These namespaces include separate namespaces for HTTP cookies based on the domain of the cookie, separate namespaces for web page instances based on the origin of the page, separate namespaces for each of our subsystems, and a separate namespace for all traditional apps. When the IBOS kernel passes a request to the storage manager it will append the security labels, a copy of the key from the key-value pair, and a hash of the contents to the payload before encrypting the data and passing it to the storage subsystem. When the IBOS kernel retrieves this data, it can decrypt the data and check the labels and integrity of the information. By using encryption, the IBOS kernel does not need to implement security invariants for any of our storage drivers, and our storage subsystem is free to make data persistent using any mechanisms it sees fit, such as the network (like in our implementation) or via a disk-based storage system.

Our current implementation does not make any efforts to avoid an attacker that deletes objects or replays old storage data. For web applications this limitation has only a small effect because the cookie standards do *not* require browsers to keep cookies persistently and because web applications often limit the lifetime of cookies using expiration dates, which are also part of the cookie standard. However, if this limitation did become problematic, we could apply the principles learned from distributed or secure file systems to provide stronger guarantees.

4.7 Network process invariants

Our IBOS kernel maintains five main invariants for network processes:

SI 4: *The kernel must route network requests from web page instances to the proper network process.*

SI 5: *The kernel must route Ethernet frames from the NIC to the proper network processes.*

SI 6: *Ethernet frames from network processes to the NIC must have an IP address and TCP port that matches the origin of the network process.*

SI 7: *HTTP data from network processes to web page instances must adhere to the SOP.*

SI 8: *Network processes for different web page instances must remain isolated.*

To help enforce these invariants, IBOS puts all network processes in their own protection domains. If a web page instance makes a HTTP request, the kernel will extract the origin from the request message and either route this request to an existing network process that has the same label, or it will create a new network process and label the network process with the origin of the HTTP request. Likewise, the kernel inspects incoming Ethernet frames to extract the origin and TCP port information, and routes these frames to the appropriately labeled network process. By putting network processes in their own protection domains, the kernel naturally ensures that network requests from web page instances and Ethernet frames from the NIC are routed to the correct network process (SI 4) (SI 5).

To ensure that the NIC sends outgoing Ethernet frames to the correct host, the IBOS kernel checks all outgoing Ethernet frames before sending them to the NIC to check the IP address and TCP port against the label of the sending network process (SI 6). Also, the IBOS kernel checks cookies before passing them to the network process to ensure that all of the origin labels adhere to cookie standards. By performing these checks, the IBOS kernel ensures that the NIC sends outgoing network requests to the proper host and that the request can only include data that would be available to the server anyway.

To enforce the SOP, the IBOS kernel inspects HTTP data before forwarding it to the appropriate web page instance and drops any HTML documents from different origins (SI 7). To inspect data, the kernel uses the content sniffing algorithm from Chrome [10] to identify HTML documents so the kernel can check to make sure that the origin of HTML documents and the origin of the web page instance match. This countermeasure prevents compromised web page instances from peering into the contents of a cross-origin HTML document, thus preventing the compromised web page instance from reading sensitive information included in the HTML document.

To help isolate web page instances from each other, we also label network processes with the origin of the web page instance (SI 8). This second label is used only for network access control decisions and does *not* affect the cookie policy, which is predicated on the origin of the network request. To access network processes, the origin of the web page instance must match the origin of this second label. By using this second label, the IBOS kernel isolates network requests from different web page instances to the same origin. As a result of this isolation, a web page instance that is served a malicious network resource (e.g., a malicious ad [41]) that compromises a network process remains isolated from other web page instances. If an attacker can compromise a network process, IBOS limits the damage to the web page instance that included the malicious content.

4.8 UI invariants

The three UI invariants that the IBOS kernel enforces are:

SI 9: *The browser chrome and web page content displays are isolated.*

SI 10: *Only the current tab can access the screen, mouse, and keyboard.*

SI 11: *The URL of the current tab is displayed to the user.*

The key mechanisms that our UI subsystem uses to provide isolation are to use a frame buffer video driver and page protections to isolate portions of the screen (SI 9). Our video driver uses a section of memory, called a frame buffer, for writing to the screen. Processes

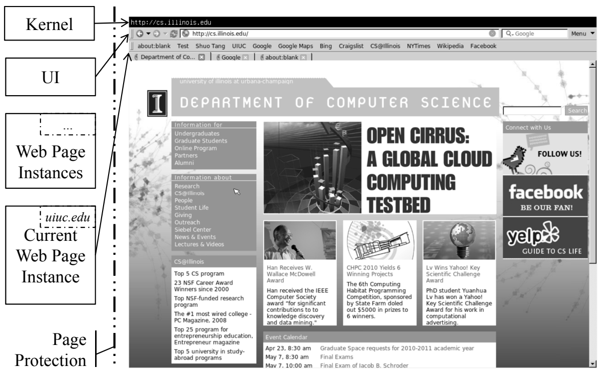
Kernel
UI
Web Page Instances
uiuc.edu
Current Web Page Instance
Page Protection
http://cs.illinois.edu
DEPARTMENT OF COMPUTER SCIENCE
OPEN CIRRUS: A GLOBAL CLOUD COMPUTING TESTBED
facebook
yelp

Figure 3: IBOS display isolation. This figure shows how IBOS divides the display into three main parts: a bar at the top for the kernel, a bar for browser chrome, and the rest for displaying web page content. The IBOS kernel enforces this isolation using page protections and *without* relying on a window manager.

write pixel values to this frame buffer and the graphics card displays these pixels. Although our mechanism makes heavy use of the software rastering available in Qt Framework[3], our experiences and anecdotal evidence from the Qt developers shows that software rastering can perform roughly as fast as native X drivers running on Linux [7]. The key advantage of our approach is that the IBOS kernel can use standard page-protection mechanisms to isolate portions of the screen. Although our current implementation does not support hardware acceleration, we believe that our techniques will work because the IBOS kernel can interpose on standardized acceleration hardware/software interfaces, such as OpenGL and DirectX.

To provide screen isolation, we divide up the screen into three horizontal portions (Figure 3). At the top, we reserve a small bar that only the IBOS kernel can access. We use the next section of the screen for the UI subsystem to draw the browser chrome. Finally, we provide the remainder of the screen to the web page instance. To ensure that only one web page instance can write to the screen at any given time, we only map the frame buffer memory region into the currently active web page instance and we only route mouse and keyboard events to this currently active web page instance (SI 10).

To switch tabs, the UI subsystem notifies the IBOS kernel about which tab is the current tab, and the IBOS kernel updates the frame buffer page table entries appropriately. However, a malicious UI manager could switch tabs arbitrarily and cause the address bar and the tab content to become out of sync (e.g., shows a page from `attacker.com`, but claims the page comes from `uiuc.edu`). One alternative we considered for this UI inconsistency was interposing on mouse and keyboard clicks to infer which tab the user clicked on, and also performing optical character recognition on the address bar to determine the address that the UI manager is displaying. However, tracking this level of detail would require far too much implementation specific information and would require the IBOS kernel to track additional events like a user switching the order of tabs.

Our approach for the IBOS kernel is to use the kernel display area to display the URL for the currently visible web page instance (SI 11). The kernel derives the URL from the label of the currently visible web page instance, providing high assurance that the URL the kernel displays matches the URL of the visible web page instance without tracking implementation specific states and events in the UI manager. Although this security invariant appears simple, it is something that modern web browsers have had trouble getting right [13].

4.9 Web page instances and `iframes`

The IBOS kernel creates a new web page instance each time a user clicks on a link or types a new URL in the address bar. To enforce the SOP on `iframes`, we run cross-origin `iframes` in separate web page instances. This separation allows us to fully track the SOP using kernel visible entities. To facilitate communication between web page instances and the `iframes` that they host, we marshal `postMessage` calls between the two.

Our current display isolation primitives are coarse grained and we rely on the web page instance to manage cross-origin `iframe` displays even though `iframes` run in separate protection domains. However, current display policies allow web page instances to draw over cross-origin `iframes` that they host, so this design decision has no impact on current browser policies. One potential shortcoming of this display management approach is that compromised web page instances can read the display data for embedded `iframes`. Fortunately, many sites with sensitive information, like `facebook.com` and `gmail.com`, use frame busting techniques [34] to prevent cross-origin sites from embedding them, which the IBOS kernel can enforce.

4.10 Custom policies

Our main focus of this project is being able to enforce current browser policies from the lowest layer of software. However, we also want to create an architecture that exposes enough browser states and events to enable novel browser security policies. Attacks such as XSS operate within traditional browser policies and can be difficult to prevent without relying on the HTML or JavaScript engine implementations. Although our architecture cannot prevent XSS, our goal is to prevent these types of attacks from causing damage.

One mechanism we implement in IBOS is to give a web server the ability to create its own more restrictive security policy to prevent attacks from sending sensitive information to third-party hosts. In our custom policy, we allow web sites to specify a server-side policy file that IBOS retrieves to restrict network accesses for a web page instance, similar to Tahoma manifests [15]. For example, assume that a bank website located at `http://www.bank.com` creates a policy file at `http://www.bank.com/.policy` that specifies the online bank system can only access resources from `www.bank.com` or `data.bank.com`. IBOS retrieves the policy file and automatically applies a more restrictive policy for the online bank web application. This restrictive policy prevents an attacker from sending stolen information to a third-party host, providing an additional layer of protection for the web application.

5 Implementation

The implementation of IBOS is divided into three parts: the IBOS kernel, IBOS messaging passing interfaces, and IBOS subsystems. The IBOS kernel is implemented on top of the L4Ka::Pistachio microkernel and runs on X86-64 uniprocessor and SMP platforms. We modified L4Ka to improve its support for SMP systems. The IBOS kernel schedules processes based on a static priority scheduling algorithm.

The IBOS kernel provides three basic APIs (i.e., `send()`, `recv()`, and `poll()`) to facilitate message passing. Applications use `send()` and `recv()` for communication and call `poll()` to wait for new messages. The IBOS kernel intercepts all messages and automatically extracts the semantics from them, like creating a new web page instance or forwarding cookies to network processes. Then the kernel inspects the semantics to make sure they conform to all security invariants and policies that we described in previous sections.

The IBOS subsystems implements APIs for web browsers and traditional applications. They are built on top of an IBOS-specific uClibc [6] C library, lwIP [17] TCP/IP stack and the Qt Framework [3]. The web browser also uses an IBOS-specific WebKit [5] to parse and render web pages.

To support traditional apps, we use our uClibc and Qt implementations to provide access to browser abstractions using the UNIX-like abstractions of the C runtime, and GUI support from Qt. We use a few Qt sample programs for testing and we implement one plugin. Our plugin is a PDF viewer that uses the Ghostscript PDF rendering engine with bindings for Qt.

System	LOC
IBOS	**42,044**
IBOS Kernel	8,905
L4Ka::Pistachio	33,139
Firefox on Linux	$>$ **5,684,639**
Firefox 3.5	2,171,267
GTK+ 2.18	489,502
glibc 2.11	740,314
X.Org 7.5	653,276
Linux kernel 2.6.31	1,630,280
ChromeOS	$>$ **4,407,066**
Chrome browser kernel 4.1.249	714,348
GTK+ 2.18	489,502
glibc 2.11	740,314
ChromeOS kernel & services (May 2010)	2,462,902

Table 1: Estimation of LOC of TCBs for IBOS, Firefox on Linux, and ChromeOS. LOC counts are also shown for some major components that are included in the TCB.

6 Evaluation

This section describes our evaluation of IBOS. In our evaluation, we analyze the security of IBOS by measuring the number of lines of code (LOC) in the IBOS TCB and comparing it with other systems, and by looking at recent bugs in comparable systems and counting vulnerabilities that IBOS is susceptible to. We also revisit the example attacks we discussed in the introduction, and we measure the performance.

6.1 TCB

In IBOS, our goal is to minimize the TCB for web browsers and to simplify browser-based systems. To quantitatively evaluate our effort, we count the LOC in the IBOS TCB and compare it against the TCB for Firefox and ChromeOS. IBOS supports fewer hardware architectures, platforms, device drivers and features, such as browser extensions, than Firefox running on Linux and ChromeOS. For a fair comparison, we only count source code that is used for running above Linux and on the X86-64 platform. Also, we omit all device drivers from our counts except for the drivers we implement in IBOS.

Table 1 shows the result of LOC counts in the TCB for these three systems, measured by SLOCCount [54]. For Firefox and ChromeOS, our counts are conservative because we only count the major components that make up the TCB for each system – there are likely more component that are also in the TCB for these systems. Because the IBOS TCB has only around 42K LOC, it is possible to formally verify or manually review the entire IBOS TCB. And in fact, one L4 type microkernel has already

Affected Component	**Num.**	**Prevented**
Linux kernel overall	21	20 (95%)
File system	*12*	*12 (100%)*
Network stack	*5*	*5 (100%)*
Other	*4*	*3 (75%)*
X Server	2	2 (100%)
GTK+ & glibc	5	5 (100%)
Overall	28	27 (96 %)

Table 2: OS and library vulnerabilities. This table shows the number of vulnerabilities that IBOS prevents.

been formally verified [32].

6.2 OS and library vulnerabilities

To evaluate the security impact of IBOS's reduced TCB, we obtained a list of 74 vulnerabilities found in the Linux kernel, X Server, GTK+, and glibc this year so far (as of Sep. 18, 2010) [1] to see how the IBOS architecture handles them. Out of the 74 vulnerabilities, 20 are related to unsupported hardware architectures and devices, and 26 cause denial-of-service, which is out-of-scope for this paper. For the remaining 28, we classify them based on the subsystem the vulnerability lies in to determine if IBOS is susceptible to these vulnerabilities.

Table 2 shows IBOS is able to prevent 27 of 28 vulnerabilities (96%). The only vulnerability we miss is a memory corruption vulnerability in the e1000 Ethernet driver. Normally IBOS is *not* susceptible to bugs in device drivers, but this particular bug resulted from the driver not accounting properly for Ethernet frames larger than 1500 bytes, and this type of logic is what our NIC verification state machine uses, so we counted this bug against IBOS.

6.3 Browser vulnerabilities

To evaluate security improvements that IBOS makes for browsers themselves, we compared how well IBOS could contain or prevent vulnerabilities found in Google's Chrome browser. For this evaluation, we obtained a list of 295 publicly visible bugs with the "security" label in Chrome's bug tracker. Out of the 295 bugs, 42 cause denial-of-service such as a simple crash or 100% CPU utilization. IBOS does not address denial-of-service or resource management currently. An additional 78 are either invalid, duplicate, not actually security issues, or related to features that IBOS does not have, such as browser extensions. For the remaining 175 bugs, we examined each of them to the best of our knowledge and classified them into the following seven categories and compared how Chrome and IBOS handle those cases:

Memory exploitation: an attacker could use a memory corruption bug to deploy a remote code execution attack. For Chrome, if the bug is in its rendering engine, Chrome contains the attack. However, bugs in the browser kernel give attackers access to the entire browser. For IBOS, bugs in either the rendering engine or other service components are contained as they are all out of the TCB.

XSS: browsers rely on careful sanitization and correct processing of different encodings to prevent XSS attacks. For both Chrome and IBOS, it is infeasible to eliminate XSS attacks, but they both contain the attacks in the affected web apps.

SOP circumvention: Chrome runs contents in frames from different origins in a single address space and uses scattered "if" and "else" statements to enforce the same-origin policy. This logic can be sometime subverted. In IBOS, we run iframes in different web page instances to provide strong isolation and check cross-origin access in the IBOS kernel.

Sandbox bypassing: Chrome uses sandboxing techniques, such as SELinux, to limit the rendering engine's authority. However, rule-based sandboxing is complex and can be bypassed in some scenarios. In IBOS, we designed browser abstractions to restrict the authority of each subsystem, which are immune to this kind of problem naturally.

Interface spoofing: browsers are sometime vulnerable to visual attacks in which a malicious website can use complex HTTP redirection or even replicate the "look and feel" of victim websites to deploy phishing. Chrome uses a blacklist-based filter to warn users of malicious websites. In IBOS, the IBOS kernel separates the display of different web page instances and uses the labels of web page instances to display the correct URL in the top of the screen to give the user a visual cue of which website he or she is visiting.

UI design flaw: some security concerns arise because of careless implementation, such as showing users' passwords in plain text. Both Chrome and IBOS are vulnerable to this type of problem.

Misc: some vulnerabilities could not easily be classified and mostly have low security severity. This is the category for those remaining bugs.

In Table 3, we show the detailed results of the analysis of the 175 vulnerabilities, broken down by the classifications above. We examined each of them to determine whether Chrome contains the threats in the affected components, and whether IBOS contains or eliminates the attacks. The table shows IBOS successfully protects users from 135 of the 175 vulnerabilities (77%).

The largest portion of bugs are browser implementation flaws that cause memory corruption and allow remote code execution. Chrome does a fairly good job containing most of them when they are in the rendering

			Chrome	IBOS
Category	**Example**	**Num.**	**Contained**	**Contained or eliminated**
Memory exploitation	A bug in layout engine leads to remote code execution	82	71 (86%)	79 (96%)
XSS	XSS issue due to the lack of support for ISO-2022-KR	14	12 (87%)	14 (100%)
SOP circumvention	XMLHttpRequest allows loading from another origin	21	0 (0%)	21 (100%)
Sandbox bypassing	Sandbox bypassing due to directory traversal	12	0 (0%)	12 (100%)
Interface spoofing	Two pages merge together in certain situation	6	0 (0%)	6 (100%)
UI design flaw	Plain-text information leak due to autosuggest	17	0 (0%)	0 (0%)
Misc	Geolocation events fire after document deletion	22	0 (0%)	3 (14%)
Overall		175	83 (46%)	135 (77%)

Table 3: Browser vulnerabilities. This table shows the number of Chrome vulnerabilities that Chrome itself contains and IBOS contains or eliminates.

engine. However, Chrome is unable to contain exploits in the browser kernel. A good example is a bug in the HTTP chunked encoding module in the browser kernel, which opens the possibility for a remote attacker to inject code. In IBOS, the TCP/IP and HTTP stack is pushed out of the TCB, and is replicated and isolated according to browser security policies. Thus, IBOS is able to contain this bug. The three memory corruption bugs IBOS could not contain were from bugs in Chrome's message passing system. Because the IBOS message passing logic resides within our TCB, we counted these bugs as bugs that IBOS would have missed.

6.4 Motivation revisited

In the introduction, we listed some examples of attacks that an attacker can use to still cause damage to modern secure web browsers by exploiting code in their TCB. We revisit these examples again to argue that IBOS can prevent them.

A compromised Ethernet driver cannot access the DMA buffers used by the device. Even if an attacker exploits the Ethernet driver, he or she still cannot tamper with network packets because the driver does not have access to DMA buffers and because the IBOS kernel validates all transmit and receive buffers that the driver sets.

A compromised storage module has little impact on data confidentiality and integrity. The IBOS kernel encrypts all data with secret keys that only the IBOS kernel has access to. Stored objects are tagged with a hash and origin information so that the IBOS kernel is able to detect tampered data. The only thing a compromised storage module can do is delete objects.

A compromised network stack is constrained as well. In IBOS, every network process runs a complete network stack. A compromised network process cannot send users' data to a third party host as the IBOS kernel ensures it can only communicate with the expected host. Network processes do have the ability to modify or replay HTTP requests, but the web server might have a mechanism to defend against replay attacks.

A Compromised window manager cannot affect other subsystems in IBOS. In IBOS, the role of window manager is simplified to only draw the browser chrome. It can change some potentially sensitive information, such web page titles. However, the IBOS kernel displays the URL of the current tab in the kernel display area, providing users with some visual cues as to the provenance of the displayed web content.

6.5 Performance

To evaluate the performance implication of IBOS's architecture, we compare its browsing experience to other web browsers running in Linux. All experiments were carried out on a 2.33GHz Intel Core 2 Quad CPU Q8200 with 4GB of memory, a 320GB 7200RPM Seagate ST3320613 SATA hard drive and an Intel PRO/1000 NIC connected to 1000 Mbps Ethernet. For Linux, we used Ubuntu 9.10 with kernel version 2.6.31-16-generic (x86-64).

We use page load latency to represent browsing experience. Page load latency is defined as the elapsed time between initial URL request and the DOM `onload` event. We compare IBOS with Firefox 3.5.9, Chrome for Linux 4.1.249. We also ported most of the IBOS browser components to Linux platform (noted as IBOS-Linux) to focus on the performance impact of our IBOS kernel architecture. In IBOS, we statically allocate processors for subsystems as follows: the kernel and device drivers run on CPU0, network processes run on CPU1, web page instances run on CPU2, and all other components run on CPU3. IBOS, IBOS-Linux, and Chrome all use a same version of WebKit from February 2010 with just-in-time JavaScript compilation and HTTP pipelining enabled. For the WebKit-based browsers, we instrument them to measure the time in between the initial URL request and the DOM `onload` event. For Firefox, we use an extension that measures these same events. To reduce noise introduced by our network connection, we load

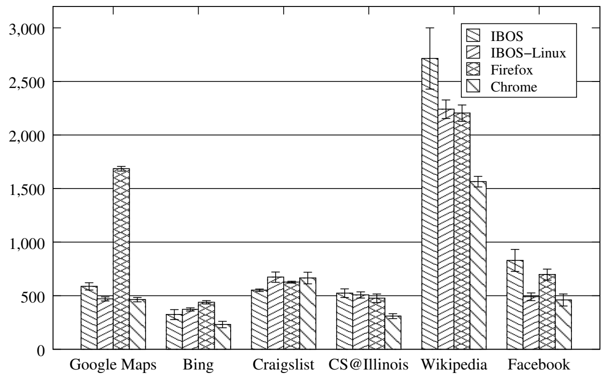

Figure 4: Page load latencies for IBOS and other web browsers. All latencies are shown in milliseconds.

each web site using a fresh web page/browser instance with an empty cache 15 times and report the average of the five shortest page load latency times.

In Figure 4, we present the page load latency times for six popular websites and show the standard deviations with the error bars. Overall, Chrome has the shortest page load latencies due to its effective optimization techniques. For `maps.google.com`, IBOS, IBOS-Linux, and Chrome out-perform Firefox, possibly due to optimization in the WebKit engine for this particular site. For `www.bing.com`, `sfbay.craigslist.org` and `cs.illinois.edu`, IBOS, IBOS-Linux, and Firefox show roughly the same results. IBOS has the fastest loading time for `craigslist. Craigslist` is a simple web site with few HTTP requests and with a large number of HTML elements. We hypothesize that the small performance improvement is due to the simplified IBOS software stack.

Both `en.wikipedia.org/wiki/Main_Page` and `www.facebook.com` have more HTTP requests than any of the other sites, and we observe slower page load latencies for IBOS than for other browsers. For these experiments IBOS performs slower than IBOS-Linux. Because we use the IBOS components in Linux, we believe that this performance difference occurs from overhead in the IBOS kernel. To test this hypothesis, we ran a number of micro benchmarks on the two systems and we believe that the overhead is due to contention for spinlocks in the L4 IPC implementation. The net effect of this contention is that heavy use of network processes requires heavy use of IPC, which adds latency to all IPC messages and slows down the overall system. However, the IBOS-Linux results for these experiments show that this slow down is not fundamental and can be fixed with a more mature kernel implementation.

Overall, the page load latency experiments show that even with a prototype implementation of IBOS, our architecture will not slow down the browsing speed significantly for the web sites we tested.

7 Additional related work

7.1 Alternative kernel architectures

Operating systems designed to reduce the trusted computing base for applications are not new. For example, several recent OSes propose using information flow to allow applications to specify information flow policies that are enforced by a thin kernel [18, 57, 33]; KeyKOS [12], EROS [45], and seL4 [32] provide capability support using a small kernel; and Microkernels [24, 27, 28] push typical OS components into user space. In IBOS, we apply these principles to a new application – the web browser – and include support for user interface components and window manager operations. Also, these previous approaches support general purpose security mechanisms, like information flow and capabilities, and shared resources and device drivers are part of the TCB. The IBOS security policy is specific to web browsers, and although this is less general, we can track this policy to hardware abstractions and can remove drivers and other shared components from our TCB.

Both Exokernels [19, 31] and L4 [27] rethink low-layer software abstractions. In both projects, they advocate exposing abstractions that are close to the underlying hardware to enable applications to customize for improved performance. In IBOS we build on these previous works – in fact we use the L4Ka::Pistachio L4 [8] MMU abstractions and message passing implementation directly. However, the key difference between our work and L4 and Exokernel is that we expose high-level application abstractions at our lowest layer of software, not low-level hardware abstractions. Our focus is on making web browsers more secure and the system software we use to accomplish this improved security.

7.2 Browser security

A number of recent papers have proposed new browser architectures including SubOS [29, 30], safe web programs [44], OP [26], Chrome [11, 43], Gazelle [52], and ServiceOS [38]. Although the browser portion of IBOS does resemble some of these works, they all run on top of commodity OSes and include complex libraries and window managers in their TCB, something that IBOS avoids by focusing on the OS architecture of our system.

The webOS from Palm [40] and the upcoming ChromeOS from Google [25] run a web browser on top of a Linux kernel. ChromeOS includes kernel hardening using trusted boot, mandatory access controls, and sandboxing mechanisms for reducing the attack surface of their system. However, ChromeOS and IBOS have fundamentally different design philosophies. ChromeOS starts with a large and complex system and tries to remove and restrict the unused and unneeded portions of the system. In contrast, IBOS starts with a clean slate and only adds to our system functionality needed for our browser. Although our approach does require implementing from scratch low-level software and fitting device drivers to a new driver model, the end result has 2 to 3 orders of magnitude fewer lines of code in the TCB, while still retaining nearly all of the same functionality.

In the Tahoma browser [15], the authors propose using virtual machine monitors (VMMs) to enable web applications to specify code that runs on the client. Tahoma uses server-side manifests to specify the security policy for the downloaded code and the VMM enforces this security policy. Tahoma does expose a few browser abstractions from their VMM to help manage UI elements and network connections, but operates mostly on hardware-level abstractions. Because Tahoma operates on hardware-level abstractions, Tahoma is unable to provide full backwards-compatible web semantics from the VMM and more fine-grained protection for browsers, such as isolating `iframes` embedded in a web application. Also, many modern VMMs use a full-blown commodity OS in a privileged virtual machine or host OS for driver support, leaving tens of millions of lines of code in the TCB potentially.

7.3 Device driver security

Device driver security has focused on three main topics. First, several projects focus on restricting driver access to I/O ports and device access to main memory via DMA. For example, RVM uses a software-only approach to restrict DMA access of devices [55], SVA prevents the OS from accessing driver registers via memory mapped I/O through memory safety checks [16], and Mungi [35] relies on using a hardware IOMMU to limit which memory regions are accessible from devices. Second, system designers isolate drivers from the rest of the system. This isolation can be achieved by running drivers in user-mode, which has been a staple of Microkernel systems [24, 36, 28], using software to protect the OS from kernel drivers [20, 58], or by using page table protections within the OS [49, 48]. The driver security architecture in IBOS differs from these approaches because our system provides fine-grained protection for individual requests within a shared driver in addition to isolating the driver from the rest of the system.

7.4 Secure window managers

A number of recent projects have looked at reducing the TCB for window managers. For example DoPE [21] and Nitpicker [22] move widget rendering from the server to the client, leaving the server to only manage shared buffers. CMW [56], EWS [46], and TrustGraph [39] also use clients for rendering, but are able to apply capabilities and mandatory access control policies to application user-interface elements. In IBOS, we deprecate the general window notion of modern computer systems in favor of the simpler browser chrome and tab motif, allowing us to track our security policies down to the underlying graphics hardware on our system.

8 Conclusions

In this paper, we presented IBOS, an operating system and web browser co-designed to reduce drastically the trusted computing base for web browsers and to simplify browsing systems. To achieve this improvement, we built IBOS with browser abstractions as first-class OS abstractions and removed traditional shared system components and services from its TCB. With our new architecture, we showed that IBOS enforced traditional and novel security policies, and we argued that the overall system security and usability could withstand successful attacks on device drivers, browser components, or traditional applications. Our experimental results showed that IBOS added little overhead when compared to today's high-performance browsers running on fast and mature commodity operating systems.

Acknowledgment

We would like to thank Brad Chen, Steve Gribble, and Hank Levy for their feedback on our security analysis. We would also like to thank our shepherd Nickolai Zeldovich, Anthony Cozzie, and Matt Hicks who provided valuable feedback on this paper. This research was funded in part by NSF grants CNS 0834738 and CNS 0831212, grant N0014-09-1-0743 from the Office of Naval Research, AFOSR MURI grant FA9550-09-01-0539, and by a grant from the Internet Services Research Center (ISRC) of Microsoft Research.

References

[1] CVE - Common Vulnerabilities and Exposures (CVE). http://cve.mitre.org.

[2] Gecko plugin API reference. https://developer.mozilla.org/en/Gecko_Plugin_API_Reference.

[3] Qt - A Cross-platform application and UI. http://qt.nokia.com/.

[4] Symantec internet security threat report april 2010. http://www.symantec.com/business/theme.jsp?themeid=threatreport.

[5] The WebKit Open Source Project. http://webkit.org/.

[6] uClibc. http://www.uclibc.org/.

[7] Qt labs blogs: So long and thanks for the blit, 2008. http://labs.trolltech.com/blogs/2008/10/22/so-long-and-thanks-for-the-blit/.

[8] L4Ka::Pistachio microkernel, 2010. http://l4ka.org/projects/pistachio.

[9] ANDERSON, J. P. Computer security technology planning study. Tech. rep., HQ Electronic Systems Division (AFSC), October 1972. ESD-TR-73-51.

[10] BARTH, A., CABALLERO, J., AND SONG, D. Secure content sniffing for web browsers or how to stop papers from reviewing themselves. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2009).

[11] BARTH, A., JACKSON, C., REIS, C., AND THE GOOGLE CHROME TEAM. The security architecture of the chromium browser, 2008. http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[12] BOMBERGER, A. C., FRANTZ, W. S., HARDY, A. C., HARDY, N., LANDAU, C. R., AND SHAPIRO, J. S. The KeyKOS nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures* (Berkeley, CA, USA, 1992), USENIX Association, pp. 95–112.

[13] CHEN, S., MESEGUER, J., SASSE, R., WANG, H. J., AND WANG, Y.-M. A systematic approach to uncover security flaws in GUI logic. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (May 2007), pp. 71–85.

[14] CHEN, S., ROSS, D., AND WANG, Y.-M. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)* (2007), pp. 2–11.

[15] COX, R. S., HANSEN, J. G., GRIBBLE, S. D., AND LEVY, H. M. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (May 2006), pp. 350–364.

[16] CRISWELL, J., GEOFFRAY, N., AND ADVE, V. Memory safety for low-level software/hardware interactions. In *Proceedings of the Eighteenth Usenix Security Symposium* (August 2009).

[17] DUNKELS, A., WOESTENBERG, L., MANSLEY, K., AND MONOSES, J. lwIP embedded TCP/IP stack. http://savannah.nongnu.org/projects/lwip/, 2004.

[18] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), ACM, pp. 17–30.

[19] ENGLER, D. R., KAASHOEK, M. F., AND JR., J. O. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 1995 Symposium on Operating Systems Principles* (December 1995), pp. 251–266.

[20] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. Xfi: software guards for system address spaces. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 75–88.

[21] FESKE, N., AND HÄRTIG, H. DOpE - a window server for real-time and embedded systems. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium* (Washington, DC, USA, 2003), IEEE Computer Society, p. 74.

[22] FESKE, N., AND HELMUTH, C. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 85–94.

[23] GARFINKEL, T. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the 2003 Network and Distributed System Security Symposium (NDSS)* (February 2003).

[24] GOLUB, D., DEAN, R., FORIN, A., AND RASHID, R. Unix as an Application Program. In *Proceedings of the 1990 USENIX Summer Conference* (1990).

[25] GOOGLE INC. Chromium OS, 2010. http://www.chromium.org/chromium-os.

[26] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the OP web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (May 2008), pp. 402–416.

[27] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., WOLTER, J., AND SCHÖNBERG, S. The performance of μ-kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1997), ACM, pp. 66–77.

[28] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. MINIX 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev. 40*, 3 (2006), 80–89.

[29] IOANNIDIS, S., AND BELLOVIN, S. M. Building a secure web browser. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (June 2001).

[30] IOANNIDIS, S., BELLOVIN, S. M., AND SMITH, J. Sub-operating systems: A new approach to application security. In *SIGOPS European Workshop* (September 2002).

[31] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEÑO, H. M., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on exokernel systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), ACM, pp. 52–65.

[32] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: formal verification of an os kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 207–220.

[33] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. In *SOSP '07: Proceedings of twenty-first ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2007), ACM, pp. 321–334.

[34] LAWRENCE, E. Combating clickjacking with x-frame-options, March 2010. http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx.

[35] LESLIE, B., AND HEISER, G. Towards untrusted device drivers. Tech. rep., UNSW-CSE-TR-0303, 2003.

[36] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GOTZ, S. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 2004 Symposium*

on Operating Systems Design and Implementation (OSDI) (December 2004).

[37] Moshchuk, A., Bragin, T., Gribble, S. D., and Levy, H. M. A crawler-based study of spyware on the web. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)* (February 2006).

[38] Moshchuk, A., and Wang, H. J. Resource Management for Web Applications in ServiceOS. Tech. rep., Microsoft Research, May 2010.

[39] Okhravi, H., and Nicol, D. M. Trustgraph: Trusted graphics subsystem for high assurance systems. In *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 254–265.

[40] Palm Inc. webOS, 2010. `http://opensource.palm.com`.

[41] Provos, N., Mavrommatis, P., Rajab, M. A., and Monrose, F. All your iFRAMEs point to us. In *Proceedings of the 17th Usenix Security Symposium* (July 2008), pp. 1–15.

[42] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., and Modadugu, N. The ghost in the browser: Analysis of Web-based malware. In *Proceedings of the 2007 Workshop on Hot Topics in Understanding Botnets (HotBots)* (April 2007).

[43] Reis, C., and Gribble, S. D. Isolating web programs in modern browser architectures. In *Proceedings of the 2009 EuroSys conference* (2009).

[44] Reis, C., Gribble, S. D., and Levy, H. M. Architectural principles for safe web programs. In *Proceedings of the Sixth Workshop on Hot Topics in Networks (HotNets)* (November 2007).

[45] Shapiro, J. S., Smith, J. M., and Farber, D. J. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles* (New York, NY, USA, 1999), ACM, pp. 170–185.

[46] Shapiro, J. S., Vanderburgh, J., Northup, E., and Chizmadia, D. Design of the EROS trusted window system. In *Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 12–12.

[47] Singh, K., Moshchuk, A., Wang, H. J., and Lee, W. On the incoherencies in web browser access control policies. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2010).

[48] Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M. Recovering Device Drivers. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004).

[49] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM, pp. 207–222.

[50] Symantec Inc. Symantec global Internet security threat report: Trends for 2008, April 2009. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`.

[51] Tan, L., Zhang, X., Ma, X., Xiong, W., and Zhou, Y. AutoISES: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security '08)* (July-August 2008).

[52] Wang, H. J., Grier, C., Moshchuk, A., King, S. T., Choudhury, P., and Venter, H. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 2009 USENIX Security Symposium* (August 2009).

[53] Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., and King, S. Automated Web Patrol with Strider HoneyMonkeys: Finding Web sites that exploit browser vulnerabilities. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)* (February 2006).

[54] Wheeler, D. SLOCcount, 2009. `http://www.dwheeler.com/sloccount/`.

[55] Williams, D., Reynolds, P., Walsh, K., Sirer, E. G., and Schneider, F. B. Device driver safety through a reference validation mechanism. In *OSDI 08: Proceedings of the 8th symposium on operating systems design and implementation* (2008).

[56] Woodward, J. P. Security requirementes for systems high and compartemented mode workstations. Tech. rep., MITRE Corp., 1987. MTR 9992.

[57] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making information flow explicit in HiStar. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 263–278.

[58] Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., and Brewer, E. Safedrive: safe and recoverable extensions using language-based techniques. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 45–60.