Improving Desktop System Security Using Compartmentalization

by

Mohsen Zohrevandi

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2018 by the
Graduate Supervisory Committee:

Rida Bazzi, Chair
Gail-Joon Ahn
Adam Doupé
Ming Zhao

ARIZONA STATE UNIVERSITY

August 2018

ABSTRACT

Compartmentalizing access to content, be it websites accessed in a browser or documents and applications accessed outside the browser, is an established method for protecting information integrity [12, 19, 21, 60]. Compartmentalization solutions change the user experience, introduce performance overhead and provide varying degrees of security. Striking a balance between usability and security is not an easy task. If the usability aspects are neglected or sacrificed in favor of more security, the resulting solution would have a hard time being adopted by end-users. The usability is affected by factors including (1) the generality of the solution in supporting various applications, (2) the type of changes required, (3) the performance overhead introduced by the solution, and (4) how much the user experience is preserved. The security is affected by factors including (1) the attack surface of the compartmentalization mechanism, and (2) the security decisions offloaded to the user. This dissertation evaluates existing solutions based on the above factors and presents two novel compartmentalization solutions that are arguably more practical than their existing counterparts.

The first solution, called FlexICon, is an attractive alternative in the design space of compartmentalization solutions on the desktop. FlexICon allows for the creation of a large number of containers with small memory footprint and low disk overhead. This is achieved by using lightweight virtualization based on Linux namespaces. FlexICon uses two mechanisms to reduce user mistakes: 1) a trusted file dialog for selecting files for opening and launching it in the appropriate containers, and 2) a secure URL redirection mechanism that detects the user's intent and opens the URL in the proper container. FlexICon also provides a language to specify the access constraints that should be enforced by various containers.

The second solution called Auto-FBI, deals with web-based attacks by creating multiple instances of the browser and providing mechanisms for switching between the browser

instances. The prototype implementation for Firefox and Chrome uses system call interposition to control the browser's network access. Auto-FBI can be ported to other platforms easily due to simple design and the ubiquity of system call interposition methods on all major desktop platforms.

To my family

ACKNOWLEDGMENTS

I like to thank my advisor, members of my graduate supervisory committee and my family and friends for their guidance, help and support during my PhD study at Arizona State University.

I am indebted to my advisor Dr Rida Bazzi for his guidance, support and encouragement for my work. I learned how to approach a challenging research problem and how to communicate academic research effectively from Dr Bazzi. I also had the opportunity to work with him as a teaching assistant for a number of years. I learned how to stay motivated in class from him and I developed a deeper passion for teaching after witnessing his tireless efforts to improve the course every semester.

I would like to thank my graduate supervisory committee members, Dr Adam Doupé, Dr Gail-Joon Ahn and Dr Ming Zhao for their insightful advice and suggestions on my work. I appreciate their time and effort in serving on my committee.

Finally, I am grateful to my family for supporting me during these years that I have been far away from home. They have always encouraged me to follow my passion and made a lot of sacrifices to support me.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER

1 INTRODUCTION 1

1.1 Contributions 5

1.2 Organization 6

2 RELATED WORK 7

2.1 Web-specific Solutions 7

2.2 Compartmentalization Solutions 8

2.2.1 Application Compartmentalization 8

2.2.2 Content-based Compartmentalization 9

2.2.3 System Compartmentalization 10

2.3 Isolation Techniques 11

2.4 Secure Graphical User Interfaces 11

2.5 Access Control 12

2.5.1 Usable Access Control 12

2.5.2 Capability Systems 13

3 SECURING ACCESS TO SENSITIVE CONTENT ON THE WEB 14

3.1 Existing Approaches 14

3.2 An Alternative Approach 15

3.3 System & Threat Model 18

3.3.1 System Model 18

3.3.2 Threat Model 19

CHAPTER Page

3.4 Design & Implementation 21
3.4.1 Design Alternatives 21
3.4.2 System Design 22
3.4.2.1 The Enforcer 22
3.4.2.2 The Enabler 25
3.4.3 Implementation Details 25
3.4.3.1 Enforcer 26
3.4.3.2 Browser Extension 29
3.4.3.3 Enabler Daemon 29
3.4.3.4 Browser Instances and User Profiles 30
3.4.3.5 Configuration Syntax 30
3.5 Performance Evaluation 33
3.6 Security Evaluation 35
3.7 Limitations 36
4 COOPERATIVE CONTAINERS FOR DESKTOP SYSTEMS 37
4.1 Existing Approaches 37
4.1.1 Inefficacy of Traditional Methods 38
4.1.2 System Compartmentalization 38
4.1.2.1 Qubes 39
4.1.2.2 Bromium 39
4.2 An Alternative Approach 40
4.2.1 Overview of FlexICon 41
4.2.1.1 Web-based Containers 42
4.2.1.2 The User Container 43

4.2.1.3 The Junk Container ... 44
4.2.1.4 The Email Container ... 44
4.2.2 Policy ... 45
4.3 Threat Model ... 48
4.3.1 Scope ... 48
4.3.2 Security Goals ... 49
4.4 Design & Implementation ... 49
4.4.1 System Design ... 49
4.4.1.1 Controller Module ... 50
4.4.1.2 Container Module ... 51
4.4.1.3 Network Module ... 51
4.4.1.4 Trusted File Dialog ... 51
4.4.1.5 URL Handling Module ... 52
4.4.1.6 GUI Module ... 52
4.4.2 System Implementation ... 52
4.4.2.1 Container Networking ... 55
4.4.2.2 Configuration Syntax ... 61
4.5 Security Evaluation ... 65
4.5.1 Attack Surface & the TCB ... 65
4.5.2 Network Attacks ... 67
4.5.2.1 Subverting the Network Filtering through DNS ... 67
4.5.2.2 Subverting the Network Filtering through Proxy ... 68
4.6 Limitations ... 69
5 COMPARING SYSTEM COMPARTMENTALIZATION SOLUTIONS ... 70

CHAPTER Page
5.1 Test Setup 70
5.2 Maximum Number of Containers 71
5.2.1 FlexICon's Memory Usage 72
5.3 Container Load Delay 73
5.3.1 Loading Containers One at a Time 73
5.3.2 Loading Containers Concurrently 75
5.3.3 Browser Launch Delay 75
5.4 Disk Access Overhead 77
5.4.1 Disk Read Throughput 78
5.4.2 Disk Write Speed 79
5.4.3 Copying Files Between Containers 82
5.5 Network Access Overhead 83
5.5.1 Network Latency 84
5.5.2 Network Throughput 84
5.6 GUI 87
6 CONCLUSION & FUTURE WORK 88
6.1 Future Work 88
REFERENCES 90
APPENDIX
A SAMPLE FLEXICON POLICY 97

LIST OF TABLES

Table Page

1. Comparing FlexICon, Qubes and Bromium 4
2. Auto-FBI Page Load Time Overhead 34
3. Maximum Number of Containers 72
4. Network Throughput 87

LIST OF FIGURES

Figure Page

1. Auto-FBI Handling a Request in Novice User Mode 17
2. Example Execution Scenario for the Enforcer 24
3. Auto-FBI System Architecture 26
4. Example Auto-FBI Configuration File Prepared by an Experienced User 32
5. Flow Graph Nodes and Edges for a Single Container 47
6. FlexICon's System Architecture 50
7. Adapted Qubes GUI Module 54
8. Example Window Border Colors 54
9. Resolving a DNS Name 56
10. Successful IP Connection 59
11. Connecting to an Unlisted IP Address 60
12. Sample FlexICon Container Definition 61
13. Sample FlexICon Rule Definitions 63
14. Avoiding Repetition with Definitions 64
15. Loading Containers One at a Time 74
16. Start Delay of Xterm in a Pre-Loaded Container 74
17. Loading Containers Concurrently 76
18. Browser Start Delay 77
19. Disk Read Overhead 78
20. Box Plot of Disk Read Throughput Data 79
21. Disk Write Overhead 80
22. Box Plot of Disk Write Speed Data for Tiny Files 80
23. Box Plot of Disk Write Speed Data for Small Files 81

Figure Page

24. Box Plot of Disk Write Speed Data for Medium Files 81

25. Box Plot of Disk Write Speed Data for Large Files 82

26. Copying Files between Containers 83

27. Latency Overhead for Serial Downloads 85

28. Latency Overhead for 2 Concurrent Downloads 85

29. Latency Overhead for 4 Concurrent Downloads 86

30. Latency Overhead for 8 Concurrent Downloads 86

Chapter 1

INTRODUCTION

Websites are vulnerable to Cross-Site Request Forgery (CSRF) [8], Cross-Site Scripting (XSS) [44] and Clickjacking [5] attacks [55]. There are specific solutions for each of these attacks, however, the solutions must be implemented by each website separately and are often tricky to implement correctly [77]. A major concern for users is that they do not know if adequate security measures are implemented for a website while they need to trust websites with sensitive information.

This concern is further compounded by the fact that the user can download content that might be malicious or the websites themselves might install content on the user's machine without the user's knowledge. Once this content executes on the user's machine, it can access user data without explicit user permission due to the access control model used in desktop systems. In all major desktop operating systems, a program is executed on behalf of the user who launches it inheriting all access privileges afforded to that user. Most of the time, the broad access privileges available to programs are not necessary and can cause security problems [1, 53, 62, 75, 76].

Researchers have proposed approaches to dealing with these concerns. Approaches that are restricted to combating CSRF, XSS and Clickjacking attacks include [8, 19, 21, 23, 39, 44, 57, 72, 73]. The existing solutions either need to be implemented by each website separately [8, 23, 39, 44, 57, 72, 73] or require websites to opt in and entail major changes in the browser code base [19, 21].

Other works [12, 60] propose compartmentalization to reduce the harm that can result from malicious content. The central idea is to break down the monolithic execution envi-

ronment of the operating system into multiple isolated execution environments (henceforth called *containers*) with limited access to user data and other system resources. If an attacker manages to run malicious code in one of these containers, the rest of the system is safe from damage. Although the key idea of system compartmentalization is simple, an actual solution requires making trade-offs between security and usability. The existing solutions, namely Qubes OS [60] and Bromium [12], represent two distinct points in this design space.

Bromium is a commercial system for Microsoft Windows based on the Xen hypervisor that supports the creation of light-weight virtual machines (called MicroVMs) for specific applications. Qubes is an open source project for providing virtual machines on top of the Xen hypervisor.

Bromium can support isolation with little effect on the user experience for a handful of commonly-used applications. For the web browser, Bromium provides isolation at the tab level and the user has essentially the same browsing experience. The user experience is slightly affected when editing and viewing documents (for instance if two Word documents are in different virtual machines, one cannot use the *switch windows* option to switch between the two documents). This ability to minimally affect the user experience is achieved by modifying the applications. Qubes on the other hand does not modify the applications and if one wishes two tabs to be isolated, one would need two separate instances of the browser in different containers (called AppVMs). Qubes requires the user to decide in which container particular content should be viewed whereas, for supported applications, Bromium relieves the user from making such decisions (for the most part). Qubes provides the ability to color the border of the application window with a color associated with its container, but the user is ultimately responsible for keeping track of the various containers. The advantages of Bromium come at a cost. Qubes disk access overhead and network over-

head are better than those of Bromium. On the other hand, Bromium can support hundreds of isolation containers whereas Qubes can support a little more than a dozen containers, but Qubes' security is arguably better than that of Bromium[1]. A solution with low performance overhead which does not modify applications and can compartmentalize all user applications must be possible.

This dissertation presents two approaches to dealing with attacks on desktop systems that are enabled by monolithic execution environments, including web-based attacks such as CSRF, XSS and Clickjacking and attacks outside the browser that rely on executing malicious code on the victim's machine. One approach is limited in scope but is easily portable to all major desktop platforms, the other approach is more comprehensive and presents a novel design in the space of system compartmentalization solutions.

The first approach, called Auto-FBI, deals with web-based attacks by creating multiple instances of the browser and providing mechanisms for switching between the browser instances. A prototype implementation of this design is presented for Firefox and Chrome on Ubuntu which uses system call interposition techniques to control the browser's network access and enforce a Chinese wall policy [11] on websites that can be accessed by browser instances. Auto-FBI can be ported to other platforms easily due to simple design and the ubiquity of system call interposition methods on all major desktop platforms.

The second approach, called FlexICon, presents an attractive alternative in the design space of compartmentalization solutions on the desktop. The main design goals for FlexICon are: (1) no modification to applications, (2) high performance, (3) reducing the effects of user mistakes, and (4) security. The first design goal means that, like Qubes, FlexICon would modify the user experience, in particular the browsing experience. Unlike Qubes,

[1]It should be noted that some of the overhead in Bromium can be due to other functionality it provides for monitoring purposes

Table 1. Comparing FlexICon, Qubes and Bromium

Aspect	FlexICon	Qubes	Bromium
Large number of containers	✓	✗	✓
Fast container launch	✓	✗	?
Fast application launch	✓	✓	✓
Low disk overhead	✓	✗	✗
Low network overhead	✓	✓	✗
Cooperative	✓	✗	N/A
Mitigates human error	✓	✗	✓
Requires no application modification	✓	✓	✗
Minimal change to user experience	✗	✗	✓

Platform	Attack Surface
FlexICon	Full kernel API
Qubes	Hypervisor + disaggregated restricted kernel API
Bromium	Hypervisor + restricted kernel API

FlexICon allows for the creation of a large number of containers with small memory footprint. The disk and network overhead of FlexICon are better than those of Qubes and Bromium. This is achieved by using lightweight virtualization based on Linux namespaces. To reduce the effects of user mistakes and the effects of the design on the browsing experience, FlexICon uses two mechanisms: 1) A trusted file dialog that allows applications running in any container to select a file for opening and launching it in the appropriate container, and 2) it provides seamless transitions between browser instances in various containers using a browser extension that reports all user-requested URLs to the system so that if the URL must be handled by a browser running in a different container, the system can

detect user's intent and open the URL in the proper container. FlexICon also provides a language that allows a savvy user to specify the access constraints that should be enforced by various containers. A summary of the comparison of FlexICon to Qubes and Bromium is shown in Table 1.

Overall, FlexICon has better performance compared to Qubes and Bromium, is more general compared to Bromium in terms of application support, makes less changes to user experience compared to Qubes, reduces error-prone security decisions by the user at runtime compared to Qubes, but has a larger attack surface compared to Qubes and Bromium.

1.1 Contributions

This dissertation has the following contributions:

(a) It presents Auto-FBI, an easily portable solution for isolating access to sensitive websites that does not require changing the browser code base or cooperation from website owners.

(b) It presents a comprehensive evaluation of existing system compartmentalization solutions in terms of performance and security.

(c) It presents FlexICon, a system compartmentalization solution that represents a new point in the design space of system compartmentalization solutions with unique tradeoffs between performance, security and usability.

(d) It presents an implementation of FlexICon for Ubuntu using light-weight isolation containers that can create large number of containers and achieve low performance overhead compared to existing systems.

1.2 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 explores the related work.
- Chapter 3 presents Auto-FBI, a limited scope solution for securing access to sensitive content on the web.
- Chapter 4 presents FlexICon, a comprehensive compartmentalization solution for desktop systems.
- Chapter 5 provides a detailed comparison of Bromium, Qubes and FlexICon.
- Chapter 6 provides conclusions and future work.

Chapter 2

RELATED WORK

This chapter explores related work in the following areas: (1) web-specific solutions, (2) compartmentalization solutions, (3) isolation techniques, (4) secure graphical user interfaces, and (5) access control.

2.1 Web-specific Solutions

Existing techniques for combating CSRF, XSS and Clickjacking attacks require each website to implement these measures individually. Content Security Policy (CSP) [72] allows the website developer to specify origins that are allowed to load in the context of a web page as well as putting restrictions on loading a page inside frames. Origin and Referer header validation [8] can be used to determine the origin of a request. SameSite cookies [73] prevent the browser from sending the cookie with cross-site requests. CSRF tokens [57, 77] can be used to mitigate CSRF attacks.

Researchers have proposed isolating web applications in a single browser to combat CSRF, XSS and Clickjacking attacks [19]. The website providers can signal the browser using the host-meta mechanism [36] to isolate their website. Implementing the isolation mechanism in the browser requires non-trivial changes to the code base of the browser which means that major browsers are not likely to adopt this method. A benefit of this approach is that it can be fine-grained beyond isolating entire origins.

Tahoma [21] provides a browser operating system (BOS), a software layer on top of which web browsers can run. The BOS employs the Xen hypervisor to compartmentalize

each web application in a Xen domain. The policy, including code to run, web domains to allow access to, and web application's identity is specified by the web provider. This approach requires major changes to the browser to be able to run on top of the BOS.

FlowFox [25] implements information flow control for web scripts based on the secure multi-execution technique. A modified version of SpiderMonkey, the JavaScript engine used in Firefox and other browsers, is used to enforce information flow security policies. FlowFox can be used to prevent information leakage by web scripts.

2.2 Compartmentalization Solutions

This section discusses the following categories of compartmentalization solutions: (1) application compartmentalization, (2) content-based compartmentalization, and (3) system compartmentalization.

2.2.1 Application Compartmentalization

Complex applications such as web browsers can be a major attack vector on desktop platforms enabling malware to find its way on the system. In order to defend against these attacks, a large complex application might be broken into multiple smaller components that execute with minimal privileges necessary to do their job. There are specific examples of applying this technique in the literature such as [7] as well as software frameworks that make it easier to apply this idea such as [71]. Although this could reduce the damage to the system in case the fortified application is attacked, ransomware and other types of malicious programs are not prevented by this method.

2.2.2 Content-based Compartmentalization

Information flow tracking and data provenance have been used to decide how to isolate processes. SPIF [68] is a system developed for Microsoft Windows that prevents untrusted applications from modifying system resources. It can prevent malware from becoming persistent on the victim's system since most malware targeting Microsoft Windows make themselves persistent across reboots by modifying certain system components. However, SPIF does not protect user's data from being modified by malware.

Shadow execution is a technique proposed by Capizzi et al. [13] to prevent desktop applications from leaking user data over the network. Two copies of the same program are executed in separate virtual machines, a private copy with no network access and a public copy with no access to user data. The responses received by the public copy over the network is then shared with the private copy to provide network functionalities such as program updates. This approach can only be used to protect confidentiality and cannot be used to protect user data against damage by malicious code. Additionally, the implementation suffers from high performance overhead.

In ServiceOS [54], applications and data are labeled based on their origin and processes are isolated based on these labels. The resulting system is more or less similar to the Android model in which every application has its own security principal. This approach requires changing applications to use a different programming model. ServiceOS also requires a custom HTTP header to be fully functional.

2.2.3 System Compartmentalization

Qubes [60] is an open source Linux-based operating system that uses Xen virtualization. Device drivers that might be vulnerable to attacks can be executed in their own dedicated virtual machines. It is also possible to create AppVMs which are user-managed virtual machine that can be dedicated to any task(s). The user needs to be vigilant to benefit from AppVMs. For example, the user needs to be constantly aware not to access unknown websites in the AppVM dedicated for online banking. Moreover, the memory overhead of full virtualization is high and booting AppVMs is slow.

Bromium [12] is a commercial product for Microsoft Windows that also employs Xen-based virtualization to create MicroVMs to isolate various tasks. A task in Bromium can be a single browser tab or a PDF viewer displaying a file. The MicroVM only provides minimal access for processes. Bromium uses patented technology to reduce memory overhead of VMs when running supported applications. Unsupported applications still cause a large memory overhead when executed in a VM. To achieve this level of isolation (for instance isolating browser tabs in separate MicroVMs) and reduce the memory overhead of full virtualization, Bromium relies on heavy modifications to applications. Additionally, the user has the following options when it comes to applications that are not supported by Bromium: (1) either trust the application with complete access to the host or (2) restrict it to a high overhead virtual environment with no access to any files on the host.

Polaris [67] is a prototype for Windows XP that allows the user to run applications in restricted mode by running them with a restricted user account. It also implements PowerBox, a trusted file dialog that can show all user files and once the user selects a file, it gives access to the file to the confined application. The confinement method used in Polaris does not prevent applications from communicating or spying on other applications.

2.3 Isolation Techniques

Virtualization and sandboxing techniques have been used to provide isolation for different purposes. Virtualization techniques can be categorized in two broad categories: (a) full virtualization such as Xen [49] and VMware ESXi [16], (b) light-weight virtualization (sometimes called OS virtualization) such as OpenVZ [56] and LXC [48]. Full virtualization systems provide the ability to run multiple heterogeneous operating systems on a single machine which is desirable for server consolidation and improving hardware utilization [69]. Light-weight virtualization systems have some limitations in that regard, for instance one cannot run Microsoft Windows in OpenVZ. However, light-weight virtualization techniques incur less overhead and offer better performance [17].

Device namespaces [3] complement Linux namespaces [10] for Android platform and provide virtualization for all device drivers. These techniques have been used to extend the Android OS to provide virtualization for smart-phones [3].

Different sandboxing methods [30, 33, 45, 70, 74] exist for desktop platforms that are used to fight malware [35] by containing unknown applications to a virtual and limited environment. Google Chrome uses sandboxing techniques to isolate its rendering engine [7]. Adobe Reader X and Internet Explorer also use sandboxing techniques for similar purposes [35].

2.4 Secure Graphical User Interfaces

Window systems serve as the mediator of user input and output and provide interprocess communication (IPC) for client applications. Widely used systems like the X window system [61] have outdated trust assumptions. They are based on the assumption that all ap-

plications are friendly and the window system should encourage and facilitate cooperation between client applications. Trusted Window Systems [26, 29, 64] try to solve the security issues that arise when these assumptions do not hold, i.e. modern computing environment. Isolation is a key principle: client applications should not be able to spy on each other. User intent traceability is also an important factor in designing a trusted window system. Preventing applications from eavesdropping on user input and preventing denial of service attacks on the GUI system are also addressed by such systems. The EROS Trusted Window System (EWS) was developed from scratch to create a new window system with security features in mind [64]. Some solutions try to be compatible with the X window system as much as possible [27, 28, 29]. The Trusted X (TX) [28] system has the additional goal of retrofitting X to provide a multi level security (MLS) scheme for the window system with minimal effects on X applications. The work also introduces a novel windows labeling scheme (on all sides) to ensure that users are aware of the window security level. Nitpicker [29] remains compatible with X applications while achieving a limited set of security goals in 1500 lines of code.

2.5 Access Control

2.5.1 Usable Access Control

The strategy of *security by designation* integrates security decisions with the user's primary task [75]. The idea is to grant authorities to programs based on user actions, e.g. when the user selects a file to open, the system would grant read access to that file to the program. Examples of applying security by designation in the literature include Access Control Gadgets for Android [59], and Polaris [67].

Access Control Gadgets (ACGs) provide user interfaces for system resources such as the camera, that grant access permission to applications that embed these interfaces when the user naturally interacts with the interface [59]. A requirement of this approach is a system that executes applications with limited privileges. Mobile platforms such as Android and iOS provide such execution models while desktop operating systems do not.

2.5.2 Capability Systems

In systems that use the *access control lists* mechanism, a process is normally executed on behalf of a user inheriting all privileges of that user. This has been described as the *ambient authority* model [53, 71]. Although it is theoretically possible to consider every process a separate subject for ACL, practical considerations dictate that the list of subjects be kept more or less static and small. On the other hand, capability systems [47, 63] use *capability lists* which allow for more fine-grained control over the privileges of a process [53]. However, capability lists require a different programming API and existing applications need to be modified in order to work on a capability system.

Capsicum [71] provides capabilities for Unix-like operating systems. It can be used to break large applications into smaller pieces that run in sandboxes and use capabilities for accessing system resources. This approach is very promising in bridging the gap between ACL-based systems and capability systems. However, only the programs that leverage the new Capsicum API benefit from it.

Chapter 3

SECURING ACCESS TO SENSITIVE CONTENT ON THE WEB

This chapter describes Auto-FBI, a novel and simple approach for securing access to sensitive content on the web. The approach automates the best manual compartmentalization practices for accessing different kinds of content with multiple browser instances. The automation is transparent to the user and does not require any modification of how non-sensitive content is accessed. For sensitive content, a Fresh Browser Instance (FBI) is automatically created to access the content. Auto-FBI can provide support for novice users with predefined sensitive-content sites as well as for more experienced users who can define conflict of interest (COI) classes which allows content from sites in the same user-defined class to coexist in a browser instance. Performance evaluation of Auto-FBI shows that the overhead introduced by the approach is acceptable (less than 160 ms for sites that already have fast load time, but for slow sites the overhead can be as high as 750 ms).

3.1 Existing Approaches

Attack vectors targeting web applications are possible due to a combination of factors: a) web standards are very permissive in terms of cross-site interactions, b) cross-site requests usually carry authenticating tokens such as session id cookies, c) determining the true origin of a request and whether or not the user knowingly initiated the request is not straightforward on the server side, d) sanitizing user inputs correctly is very tricky [77].

Existing techniques for combating CSRF, XSS and Clickjacking attacks are discussed in Section 2.1. A shortcoming of these methods (CSP, SameSite cookies, Origin and Ref-

erer header validation, CSRF tokens) is that they need to be implemented for each website separately. Moreover, the evolving nature of the web, lack of universal browser support for newer methods such as CSP and SameSite cookies, filtering of the Referer HTTP headers due to privacy concerns and other complicating factors make it difficult to reliably eliminate these attack vectors.

A common security advice for users is to use at least two browsers to surf the web, one for sensitive sites, and one for other purposes [14, 57, 66]. The main reason given for this advice is usually the following. "Using multiple browsers can minimize the chances that a vulnerability in a particular web browser, website, or related software can be used to compromise sensitive information" [14]. Following this advice in practice is error-prone and cumbersome especially for novice users, but it demonstrates the need for compartmentalization of web access.

3.2 An Alternative Approach

A simple and practical solution would be to automate the best practice of using multiple browsers by creating multiple browser instances[2] and providing mechanisms for switching between multiple browser windows to reduce the burden on the user. Auto-FBI implements this approach. It works by launching a fresh browser instance (FBI) whenever the current browser instance should not be used to access the requested content.

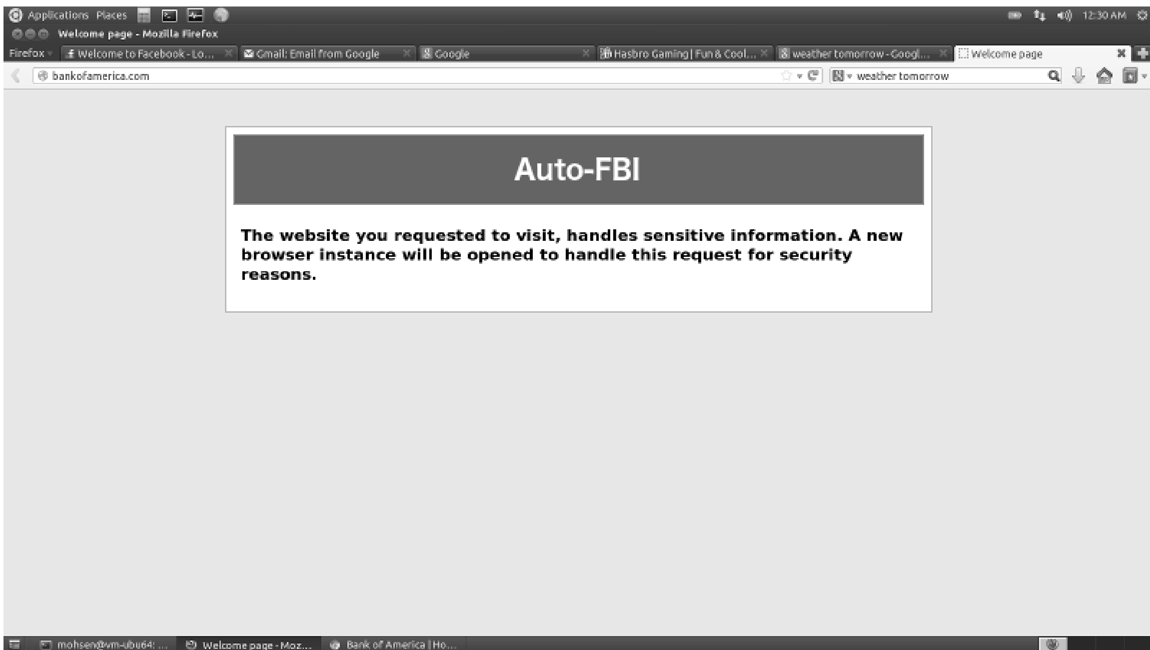
Auto-FBI provides two usage models, one for novice users and one for savvy users. The usage model for novice users is simple. In this model there are two kinds of websites: sensitive and non-sensitive. Websites are classified in one of these two classes by listing

[2] A browser instance is a combination of the browser code and state, including browsing history, cookies, cache, saved passwords, etc.

sensitive sites in a configuration file. The user normally interacts with the default browser instance that is associated with non-sensitive sites. Once the user tries to navigate to a sensitive site in the non-sensitive instance, a new browser instance is created and a new window from that browser instance opens to handle the request. To emphasize that the window is only meant for navigating one site, the address bar, menus and tabs are disabled in the new window.

Experienced users can define more than two classes of websites and can control the type of browser (Firefox or Chrome) to use in each case. Each class of websites is essentially a conflict of interest (COI) class and Auto-FBI enforces a Chinese wall policy [11] by allowing websites that belong to the same COI class to co-exist in the same browser instance but preventing websites belonging to different COI classes from co-existing in the same browser instance. To allow for infrastructure sites such as certificate authorities that need to be accessible in all browser instances, Auto-FBI allows experienced users to list such domains separately and allows all browser instances to access them.

COI classes are specified by listing websites that should be allowed to co-exist in the same browser instance. An implicit general class captures websites that are not listed in any other class. The user normally interacts with the default browser instance associated with the general COI class. If the user tries to navigate to a websites that is not allowed in the current browser instance, a new browser instance is created (if one does not exist already) and Auto-FBI opens the requested website in that browser instance and brings the window to focus. The original browser instance is prevented from loading the website. Auto-FBI redirects the request from the original browser instance to a local page showing a message to the user that the site will be opened in another browser instance. The browser extension can automatically close this page to reduce clutter if desired. Figure 1 shows an



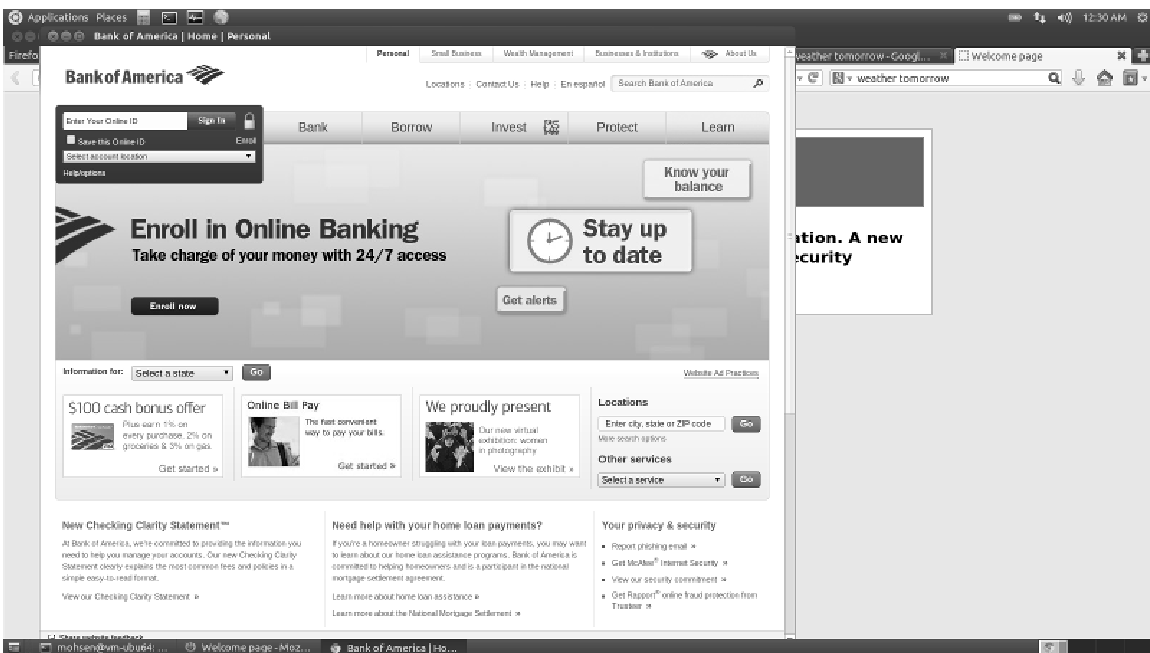


Figure 1. A request is made to `www.bankofamerica.com` in the general browser instance. Instead of displaying the requested page in the current instance, Auto-FBI displays a message (top image) and opens the requested site in a fresh browser instance (bottom image).

example where the original browser instance is showing a message from Auto-FBI and a new browser instance is loading the requested page in novice user mode.

3.3 System & Threat Model

3.3.1 System Model

This section identifies the relevant components of the system. The system consists of a client side and a server side. Web clients typically run in *web browsers*. Web browsers retrieve *web content* delivered by *web servers*. Web servers and web content have *origins* associated with them. An origin is considered a principal (whether or not it is authenticated). Web content is acted upon by the web browser. Content might be *active* which requires the browser to execute some code to process it. A browser that processes content from a given origin is considered a *delegate* of the origin. Web content is retrieved using *web queries*. A query can be a *simple query* that specifies the address of the content in the form of a URL (uniform resource locator) which consists of a *domain name* and file path or an *active query* that consists of the above and parameters that are passed to server-side code that use the parameters to process the query. A web server might require *credentials* from a web browser in order to authenticate the principal requesting queries. Credentials are typically provided when initial access is requested, typically in the form of a *user id and password*. Once credentials are approved, a *cookie* is stored by the browser to be used for future authentication. At that point the browser acts as a delegate of the authenticated principal.

An *instance of a web browser* is a software process (or a group of related processes) that executes the browser code and has associated with it a *profile* which includes information

such as preferences, browsing history, cookies, bookmarks, and saved passwords. A profile captures an *execution state* of the browser. When a browser exits, some information might be deleted from the profile, non persistent cookies for example, but other information in the profile is saved. When a browser is restarted after exiting, the profile provides continuity between the old browser process and the new process.

3.3.2 Threat Model

Given that a browser instance can be processing content from different sources, and therefore is a delegate for these sources, it is important that these various roles do not get mixed together in the browser. For example, a script running in a browser on behalf of one source should not be able in general to read data associated with another source in the browser – such as passwords being entered in a form. Such restrictions are expressed as security policies and are enforced by the browser. Unfortunately, given the great flexibility provided by the browser, scripts have the ability to obtain information indirectly through attempted access to forbidden resources and to communicate directly and indirectly with their origins. This flexibility makes it hard to enforce policies and even when policies are correctly enforced, they might still allow undesired behavior. Same Origin Policy (SOP) [6] is an example of a policy that can be circumvented rather easily [43]. In general, a *security policy* restricts what actions can be done by the various system processes. Abstractly, one can think of a policy as a *set of allowed system executions*. A policy for a web browser expresses what executions of the browser are allowed and which executions are not allowed as seen by an external observer. In other words, a policy on a web browser is a restriction on the input/output behavior of the browser.

A browser is susceptible to an *attack* relative to a security policy if some browser execu-

tions do not satisfy the security policy. The attacks are not defined in terms of the specific technical modalities to achieve them, but in terms of what can be achieved. In general, an attack requires the browser to get and process content from one or more origins. The following description of attacks assumes that there are at least three classes of origin: HIGH, LOW and AUTH (authenticated). It is also assumed that LOW and HIGH are disjoint. The following are attack models that are considered in this chapter.

1. Information Leakage (Leakage): An attacker is able to launch an information leakage attack if a LOW origin is able to receive content from a HIGH origin through the browser.
2. Cross Site Request Forgery (CSRF): An attacker is able to launch a CSRF attack if requests to an AUTH origin to which the browser is already authenticated *depends on* content from a different origin.
3. Clickjacking (UI Redress): UI redress attack is similar to CSRF attack, but requires user participation by interacting with user interface elements. It is typically achieved by exploiting features of the user interface to let a user interact with a target website when they think that they are interacting with other content.

The way the attacks are described does not necessarily follow the way they are typically described in the literature, but it is a more general description that emphasizes the attacks capabilities rather than the minutiae of the attack mechanisms.

In this chapter it is assumed that a browser cannot be compromised in a way that enables it to access system resources it is not supposed to access. For example, the browser will not arbitrarily read and write files it is not supposed to access.

3.4 Design & Implementation

The desired functionalities can be achieved using two mechanisms: an *enforcer* mechanism and an *enabler* mechanism. The enforcer mechanism prevents the browser instance from communicating with web origins that are not in its COI class. The enabler mechanism starts browser instances as needed.

3.4.1 Design Alternatives

There are three ways to design the enforcement mechanism: (a) inside the browser by modifying the browser code, (b) as a browser extension, (c) outside the browser. The following is a brief discussion of these option:

- **Enforcement inside the browser:** The desired functionality could be provided by modifying the browser to add the desired enforcement rules. This approach is highly browser-dependent and requires detailed understanding of the browser's code to be implemented. It requires different implementations for different browsers. Since this approach does not provide a clean separation between the browser's code and the enforcement mechanism, it is not recommended.
- **Enforcement in a browser extension:** Providing the desired enforcement as a browser extension is an improvement over enforcement inside the browser. In a browser like Chrome, extensions can be a combination of JavaScript and native code that is controlled by JavaScript. This approach is also not advisable because the JavaScript engine is potentially exposed to malicious scripts from various sources. Any compromise of the engine through such malicious scripts can also compromise the enforcement mechanism.

- **Enforcement outside the browser:** Enforcement outside the browser can be achieved by intercepting the browser's system calls and only allowing those calls that are according to policy. This has the advantage of separating the enforcement mechanism from the browser's code. It also obviates the need to make additional restrictive assumptions about the power of the attacker. Enforcement outside the browser does not preclude implementing part of the enabler mechanism inside the browser.

Auto-FBI follows the third design option for the enforcement mechanism. The enabler mechanism is also mainly outside the browser, however, a browser extension is used to report user-requested URLs to the enabler mechanism running outside the browser.

3.4.2 System Design

Auto-FBI has two main components: the enforcer and the enabler. Each browser instance runs inside an enforcer which controls the system calls made by the browser to ensure that the browser instance can only access websites in its COI class. The enabler determines when it is necessary to launch a new browser instance to handle a user request based on the system configuration.

3.4.2.1 The Enforcer

The enforcer uses system call interposition techniques to control the browser's network access. When the browser issues a system call, the enforcer is notified by the operating system. If the system call is not relevant to network access, the enforcer does not change anything. Otherwise, it might change values returned from the operating system. This tech-

nique is used to rewrite DNS responses when the browser instance is not allowed to access the target website domain. If a DNS response message resolves the IP address of a website that is not allowed in the controlled browser instance, the enforcer changes the IP address in the message to the loop-back IP address which prevents the browser instance from obtaining the IP address of the target website. If the controlled browser instance is allowed to access the target website, the enforcer stores the IP address found in the DNS message in a *white-list*. System call interposition is also used to prevent the browser instance from connecting to IP addresses that are not in the enforcer's white-list.

The IP addresses used by the browser are not always preceded by DNS queries. In fact for security reasons, IP addresses obtained from DNS queries are typically cached by browsers even if the time to live (TTL) field is set to zero in the DNS response (IP pinning [42]). There are three kinds of IP addresses that can be used by a browser:

- IP addresses that are cached by the browser and obtained from a previous DNS query. The enforcer stores IP addresses that are obtained through DNS queries in a white-list for websites that are allowed to be accessed in the controlled browser instance. A later use of such an IP address is compared against the white-list. If the IP address is not on the list, the connection is blocked by the enforcer.
- Browser hard-coded IP addresses. Auto-FBI does not allow the use of any hard-coded IP addresses by the browser. Hard-coded addresses are considered suspicious and a security risk.
- Hand-coded IP addresses. IP addresses that are directly entered by the user have legitimate usages such as setting up some LAN services (WiFi router setup for example). Such usages should be allowed. As a policy one can allow some LAN IP addresses and associate a COI class with each allowed address.

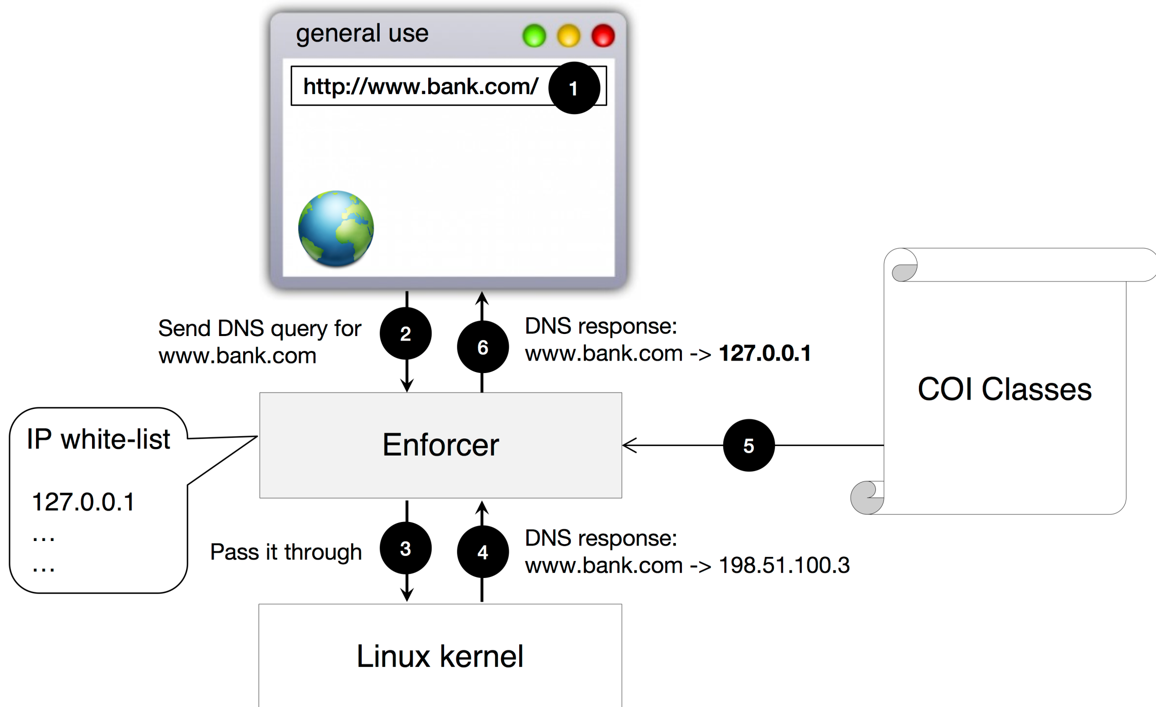


Figure 2. Execution scenario for handling access to content from a different COI class

It is interesting to note that associating a separate COI class with LAN addresses can prevent DNS rebinding attacks which are not completely solved by DNS pinning [42].

Figure 2 illustrates a scenario in which a browser instance is trying to access a website whose COI class is different from that of the browser instance. First the browser makes a DNS query to resolve the IP address (1,2,3), the DNS response is detected by the enforcer (4) and the requested domain name is checked against the list of COI classes (5). Since the browser instance is associated with a different class, the enforcer rewrites the IP address in the DNS response (6) and returns the result to the browser. The browser would not have access to the actual IP address for `www.bank.com` and hence it is not able to connect to it. Even if the browser somehow obtained the IP address for `www.bank.com`, it would not be able to connect to the website because the IP address of the website is not added to the white-list of the enforcer controlling it.

3.4.2.2 The Enabler

To manage browser instances running in the system and launching new instances, a daemon keeps track of all browser instances. To detect user-requested URLs, a browser extension in each browser instance communicates with the enabler daemon and reports all user-requested URLs to the daemon. When the enabler daemon detects a user request that cannot be satisfied in the reporting browser instance, it launches a new browser instance if needed and asks the browser extension running in the new instance to open the URL in a new tab.

Executing part of the enabler mechanism inside the browser does not affect the security of Auto-FBI. If the browser extension is compromised, all that would be affected is the URL communicated by the extension, but that has no effect on the tracing of system calls and blocking them which is enforced through a mechanism that is completely outside the browser. So, compromising the browser extension might affect the *progress* requirement of the browser, but does not affect the *safety* requirements of the enforcement.

3.4.3 Implementation Details

Auto-FBI was implemented for 64-bit Linux with browser extensions developed for Firefox and Chrome. It uses the Linux `ptrace` API [50] for system call interposition. Auto-FBI fully runs in user-space and no kernel modification is necessary. The components running outside the browser (the enforcer and the enabler daemon) are implemented in 2642 lines of C. The browser extension for Firefox is written in 133 lines of JavaScript and 139 lines of C and the browser extension for Chrome is written in 188 lines of JavaScript and 130 lines of C. Figure 3 shows the system architecture.

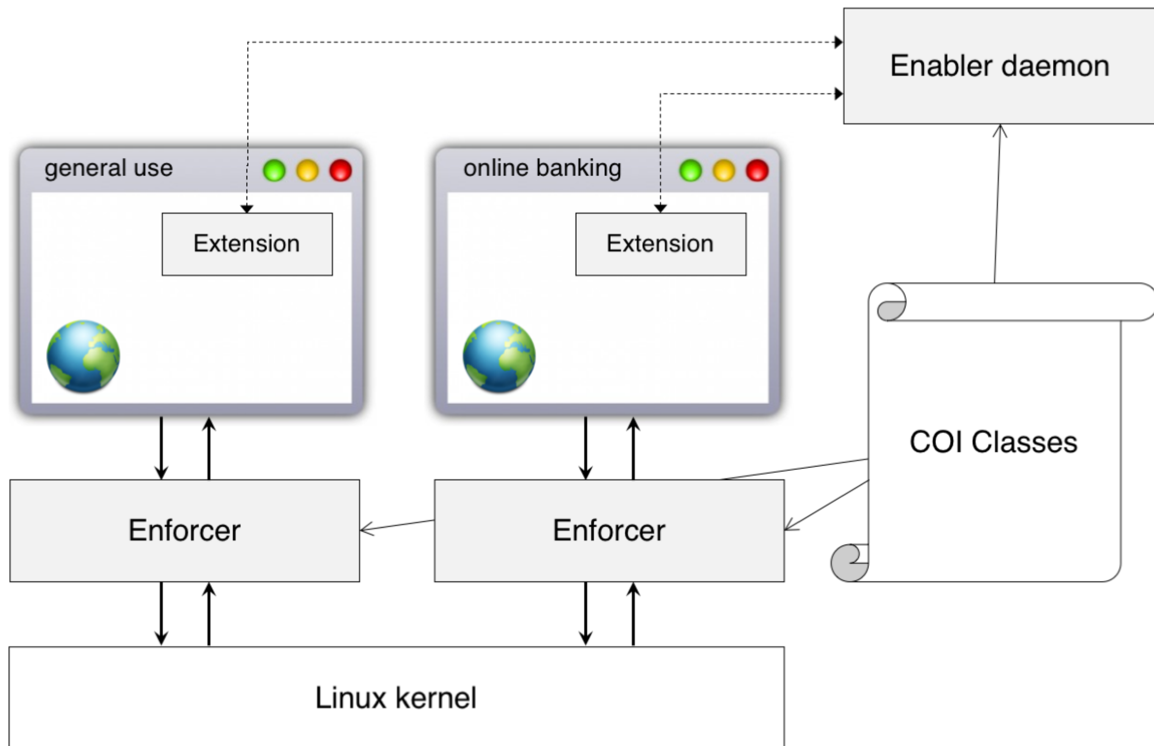


Figure 3. Auto-FBI system architecture

3.4.3.1 Enforcer

Using Linux `ptrace` API, one can trace every system call made by a child process to keep track of the child process's access to system resources including network and file system. There are two options for a tracer to start tracing the tracee: (a) `TRACEME`, (b) `ATTACH`. The first option is usually used in the following way: the tracer forks a new child process and in the child process calls `ptrace()` with `TRACEME` command, which causes the child process to be traceable by its parent process (the tracer), then the child process runs the target program using `exec()`. The second option can be used to trace an already-running process by attaching to it.

The enforcer uses the TRACEME option by default, but in order to support Chrome's use of SETUID binaries, the enforcer uses the ATTACH option for tracing Chrome[3].

Using the ATTACH method leaves an enforcement gap: from the time that tracer launches Chrome until the tracer attaches to the browser, the browser is not traced. This should not be a source of vulnerability if the default home pages are safe. Using ATTACH means that part of the enforcer that is involved in setting up the tracing is browser dependent, but the rest of the enforcement when the ATTACH option is used and the whole enforcement using the TRACEME option are generic and not dependent on the browser.

The ptrace API allows one to trace a process and all its child processes and threads by specifying the following options: PTRACE_O_TRACEFORK and PTRACE_O_TRACECLONE. Using these flags guarantees that a process cannot escape the tracer by creating a child process or a new thread.

For every system call made by the tracee, the tracer is notified twice: once before the call is handed over to the kernel for execution and once after the call returns from the kernel but before it is handed back to the tracee. So the tracer has two chances to modify a system call: it can change the parameters provided by the tracee to the kernel, or change the values returned by the kernel. To restrict the browser's access to websites, the following system calls are monitored:

- connect(): Connects a socket file descriptor to a network address.
- recvfrom(): Receives data through a socket file descriptor.

[3]This is because the Linux kernel does not allow a non-root process to trace a process with root privileges using the TRACEME option. Using TRACEME with Chrome would either require root privileges for running the browser which is not desirable or it breaks the browser's internal sandboxing mechanism which uses the SETUID flag to allow its sandbox process to temporarily run with root privileges. The enforcer waits until the privilege level is dropped after the sandbox is setup and then attaches to the main browser process using the ATTACH option of ptrace.

- `read()`: Can be used to receive data from the network (through a socket file descriptor)

The `recvfrom()` and `read()` system calls are monitored to detect DNS queries. If the port number[4] matches the port number of the DNS protocol (53), the I/O buffer from the tracee's memory is read to analyze the DNS message. If the IP addresses in the DNS message need to be changed, the tracer does so by writing to the tracee's virtual memory[5].

To prevent the browser from connecting to a restricted website by using a hard-coded IP address, the enforcer keeps a white-list of IP addresses that the browser is allowed to connect to. Initially, it is populated with the DNS service IP address (usually the local DNS client, 127.0.0.1) and a list of predefined IP addresses (to allow for LAN addresses). When the browser makes a DNS query, one of the following happens: either the response is changed to prevent access in case the URL is not allowed, or the IP addresses in the response are added to the white-list. When the browser issues a `connect()` system call, the enforcer only allows it to go through if the target IP address is in the white-list.

Note that there are other Linux system calls [51] that are similar to `read()` and `recvfrom()` in that they can be used to read from socket file descriptors, e.g. `recvmsg()`, and those system calls can be monitored similarly to detect DNS queries. However, the Auto-FBI prototype implementation does not include those system calls simply because the target browsers only used `read()` and `recvfrom()` at the time that Auto-FBI was implemented. Also note that if the tracer misses such a system call, it would not hurt security but prevent the browser from connecting to the web.

[4]The `/proc` interface was used to figure out the remote port number of a socket. Alternatively, one can keep track of all sockets owned by the tracee and the port number specified during the connect call, but this method is error-prone. See [32] for a detailed discussion on why replicating the kernel state is a bad idea.

[5]The `process_vm_readv()` and `process_vm_writev()` system calls are used to read/write data from/to the tracee's virtual memory which is much more efficient than ptrace for this purpose.

3.4.3.2 Browser Extension

The browser extension for Chrome was written using the `chrome.*` API and the browser extension for Firefox was written using the Firefox Add-on SDK. Since the JavaScript code cannot directly communicate with the enabler daemon, a native library written in C is used to forward messages between the daemon and the JavaScript code. The native library communicates at one end with the browser extension through the browser API (*Native Messaging* API in case of Chrome and *js-ctypes* in case of Firefox) and on the other end with the enabler daemon through Unix domain sockets. The native library runs as a child process of the browser instance and hence can be considered a part of the browser extension.

3.4.3.3 Enabler Daemon

The enabler daemon is a multi-threaded C program that uses Unix domain sockets to communicate with browser extensions. It is responsible for deciding how to handle restricted URLs by either dispatching it to an already-running instance that is allowed to access the URL or creating a fresh new instance.

For every browser instance running in the system, the daemon has a separate thread to communicate with the browser extension running in that browser instance. It keeps a record for each running browser instance along with a queue of outgoing messages for that instance in a shared data structure. When it receives a URL from a browser extension, it compares the COI class of the URL with that of the sender. If they match, it ignores the reported URL, otherwise it opens the URL in the proper browser instance.

3.4.3.4 Browser Instances and User Profiles

A browser instance is associated with a *profile* that captures its execution state. In practice, most browsers support *user profiles* to enable the multi-user use case. User profiles are very similar in different browsers: a folder stored somewhere in the user's home directory that contains information such as the user's web history, cookies, cache, and preferences. Fortunately, most browsers also provide a way of using multiple profiles for a user in addition to the user's default profile. In case of Firefox and Chrome, creating a fresh instance of the browser is as easy as creating an empty directory and instructing the browser to use that directory to store the user profile (though this would be cumbersome to do manually).

In Auto-FBI, the default user profile (the one managed by the user) is associated with the general COI class. Other COI classes do not have a persistent profile: a temporary profile is created whenever the instance is created and it gets deleted after the instance is closed. This way of managing user profiles is both usable and secure. User spends most of the time using the default profile since most URLs fall into the general COI category, hence allowing the user to manage that profile improves usability. Using temporary profiles for other COI classes ensures tighter security without much effect on usability.

3.4.3.5 Configuration Syntax

COI classes are specified in a single text file with the following sections separated with empty lines:

1. The first section lists domain names that should be accessible in all browser instances. This is useful for specifying certificate authorities and other infrastructure domain names.

2. Subsequent sections each specify one COI class by listing domain names of websites that belong to that COI class.

An implicit COI class, called the general COI class, captures any website that is not explicitly listed in the configuration file. Each section starts with a line that specifies the type of browser (Firefox or Chrome) that should be used when launching a new browser instance for that COI class. This line starts with `'>'` followed by a type descriptor character:

- `'F'`: use Firefox
- `'C'`: use Chrome
- `'='`: use the same browser type as the one used to make the request
- `'!'`: use a different browser type from the one used to make the request
- `'_'`: places no restriction on the browser type

The general COI class has the implicit type descriptor `'_'` which means that both Firefox and Chrome can be used with the general class. Figure 4 shows an example configuration file. Note that lines starting with # are ignored by Auto-FBI.

The example configuration file shown in Figure 4 defines two explicit COI classes: C_1 and C_2. The first class C_1 has type descriptor `'!'` and the second class has type descriptor `'C'`. To demonstrate the user experience, let's consider the following scenario: the user opens Chrome to start browsing, the browser starts in the general COI class. The user can navigate to any site that is not listed in C_1 or C_2 in this browser instance. If the user tries to navigate to `www.bankofamerica.com` which is listed in C_1, a fresh instance of Firefox will be created to handle the request. That is because the type descriptor for C_1 specifies that an instance with a different type from that of the browser used to make the request (Chrome in this example) should be used. If the user tries to access

```
# Certificate Authorities and other common domains
>
verisign.com
comodo.com
digicert.com
entrust.net
globalsign.com
sb-ssl.google.com

# C1: Banking
>!
americanexpress.com
aexp-static.com
aexp.demdex.net
bankofamerica.com
usbank.com

# C2: Shopping
>C
amazon.com
ebay.com
ebaystatic.com
ebayimg.com
```

Figure 4. Example Auto-FBI configuration file prepared by an experienced user

`www.facebook.com` from the Firefox instance, the request will be handled by the existing Chrome instance of the general class. That is because the general class has the implicit type descriptor `'_'` which allows the system to use the available instance. If the user navigates to `www.ebay.com` from any of the previous instances, a new instance of Chrome will be launched to handle the request regardless of the browser type used to initiate the request.

3.5 Performance Evaluation

To measure the impact of system call interposition on the browser, the page load time was measured using the benchmarking extension from the Chromium project [20] for 20 most-visited websites. Chromium browser version 25 was used to perform the tests. The page load time was measured when running the browser natively (T_{native}) and when running the browser with the enforcer ($T_{enforcer}$). The page load times were also measured with and without caching. Each test scenario was repeated 50 times. The page load overhead is calculated using the following formulas:

$$\Delta T = T_{enforcer} - T_{native}$$

$$\Delta T\% = \frac{\Delta T}{T_{native}} * 100$$

Table 2 shows the experiment results and calculated overheads. The last three rows of the table show the average over different sets of websites. *Average for fast sites* shows the average for those sites that have $T_{native} \leq 1000$ milliseconds and *Average for slow sites* shows the average for those sites with $T_{native} > 1000$ milliseconds. The last row shows the average over all 20 websites.

Although the relative overhead is large (about 30 %), the absolute delay is not significantly perceptible to the user. In fact for fast sites, the average overhead is less than 250 milliseconds which in absolute terms is not significantly perceptible. For slow sites, the absolute overhead is higher, but the relative overhead is lower. While the overhead of the initial prototype is acceptable, further improvements are needed. It should be noted that due to variations in data transfer rates that are unavoidable, the overhead values are negative in a few cases.

Table 2. Auto-FBI Page load time overhead, values are in milliseconds

	Caching Disabled				Caching Enabled			
Website	T_{native}	$T_{enforcer}$	ΔT	$\Delta T\%$	T_{native}	$T_{enforcer}$	ΔT	$\Delta T\%$
`www.youtube.com`	800.6	1249	448.4	56.01	696.1	1114.5	418.4	60.11
`www.yahoo.com`	953.3	890.1	-63.2	-6.63	869.7	1133.3	263.6	30.31
`login.live.com`	701.1	817.8	116.7	16.65	655.9	498.4	-157.5	-24.01
`www.msn.com`	368.9	772	403.1	109.27	367.2	613.2	246	66.99
`en.wikipedia.org`	2101.3	2199.5	98.2	4.67	823.4	1037.6	214.2	26.01
`blogsofnote.blogspot.com`	522.1	1016.7	494.6	94.73	355.8	525.9	170.1	47.81
`www.baidu.com`	1433.3	954.6	-478.7	-33.40	848.5	589.3	-259.2	-30.55
`www.microsoft.com`	1269.5	2175	905.5	71.33	1606.3	1827.9	221.6	13.80
`www.qq.com`	6430.9	8179.3	1748.4	27.19	2717	3438.5	721.5	26.56
`www.bing.com`	184.1	267.5	83.4	45.30	172.6	256.7	84.1	48.73
`www.ask.com`	952	1184.5	232.5	24.42	891.8	1034	142.2	15.95
`www.adobe.com`	1394.6	2225.9	831.3	59.61	974.8	1626.2	651.4	66.82
`www.taobao.com`	2435.7	3286.1	850.4	34.91	1802.2	2625.4	823.2	45.68
`twitter.com`	401.6	593	191.4	47.66	325.4	447.9	122.5	37.65
`www.youku.com`	6427.5	5731.2	-696.3	-10.83	2009.1	2638.5	629.4	31.33
`www.soso.com`	947.1	983.5	36.4	3.84	647.4	672.5	25.1	3.88
`wordpress.com`	601.4	799.6	198.2	32.96	370	548.9	178.9	48.35
`www.sohu.com`	5601.8	7031	1429.2	25.51	4696.7	6596.9	1900.2	40.46
`www.hao123.com`	4598.4	4600.2	1.8	0.04	2276.5	2441.5	165	7.25
`www.facebook.com`	424.9	791.7	366.8	86.33	411.7	534.7	123	29.88
Average for fast sites	623.37	851.4	**228.03**	36.58	600.74	759.51	**158.77**	26.43
Average for slow sites	3521.44	4042.53	**521.09**	14.8	2517.97	3261.45	**743.48**	29.53
Overall (20 sites)	1927.5	2287.4	**359.9**	18.67	1175.9	1510.1	**334.2**	28.42

3.6 Security Evaluation

The argument for the security of Auto-FBI is based on the fact that access to content from one COI class is not allowed in a browser instance associated with another COI class. This is enforced by restricting communication to IP addresses that are on the white-list for the COI class of the executing browser instance. The IP addresses on the white-list of one class can be one of the following: (1) IP addresses corresponding to a domain from the common domain list, or (2) IP addresses of a domain in the COI class. So, if DNS is not compromised, only content from sources in a given COI class can run on a browser instance associated with that class. For sensitive content whose servers are typically authenticated (served over HTTPS), the assumption on DNS not being compromised is not needed and the system provides the desired isolation that ensures that there is no leakage of sensitive content between various instances (assuming the authentication mechanism is secure). This means that attacks such as CSRF or Clickjacking cannot be used to access sensitive content. In fact, websites used to launch such attacks are assumed to belong to COI classes that are not part of the sensitive COI classes. This is especially evident in the novice user scenario. For the experienced user, a non-judicious choice of classes can pose a security risk.

The Auto-FBI design does not account for communication or sharing of execution state among browser instances through channels other than the network or browser profile. In particular, Local Shared Objects (LSO) can be used by Flash scripts embedded in web pages in a similar manner as HTTP cookies to communicate with other Flash scripts running in completely different browsers. In fact, an LSO can be used by websites to track users across multiple browsers or after deleting HTTP cookies [65].

System call interposition methods inherently suffer from the Time Of Check Time Of Use (TOCTOU) race conditions [33]. However, for the purpose of Auto-FBI, such race

conditions are not a major concern since the browser code is assumed to be benign and furthermore the threat model does not include threats that can compromise the browser itself.

Since the enforcer prevents the browser instance from connecting to websites outside of its COI class, attackers cannot directly launch a CSRF attack from an attacker-controlled website targeting a sensitive website in another COI class. However, they can craft a CSRF link and lure the user into clicking on the link so that the enabler would open the link in another browser instance. It is easy to prevent such attacks by sanitizing the URLs before opening. The enabler daemon can strip the query and path parts of the URL [9] before opening it in another browser instance. This URL sanitization was not implemented in the Auto-FBI prototype which is a limitation of the implementation.

3.7 Limitations

The prototype implementation of Auto-FBI has a few limitations:

1. It does not support IPv6.
2. The configuration syntax is not general. A more powerful syntax could allow wild-card expressions for specifying domain names.
3. The enabler daemon does not sanitize URLs before opening.

Chapter 4

COOPERATIVE CONTAINERS FOR DESKTOP SYSTEMS

This chapter identifies the shortcomings of Bromium and Qubes OS, the two main compartmentalization solutions for the Desktop and propose a new system, FlexICon, that presents an attractive alternative in the design space of compartmentalization solutions on the desktop. Unlike Bromium, FlexICon does not require modifying applications and is not limited to a handful of supported applications. Like Qubes, and unlike Bromium, FlexICon requires separate windows for viewing browser tabs that need to be isolated from each other. Unlike Qubes, FlexICon provides seamless transitions between browser instances in various containers. FlexICon allows the creation of containers based on online sources of content as well as containers that are not tied to online sources. At the heart of FlexICon is a mechanism that allows the definition of flexible containers with the ability to specify a rich set of access constraints to be enforced for each container. The prototype system is implemented on Ubuntu and supports the Firefox browser.

4.1 Existing Approaches

Attackers leverage the fact that programs are executed with broad access privileges. A common attack vector involves tricking users into downloading and executing seemingly useful and benign programs which are in fact malicious. Another common attack vector involves opening malicious documents that are specially crafted to exploit certain vulnerabilities in benign applications.

4.1.1 Inefficacy of Traditional Methods

Desktop operating system vendors mainly rely on the following mechanisms to address the aforementioned attack vectors: a) monitoring for suspicious behaviors through anti-virus software, b) patching discovered software vulnerabilities, c) various runtime and compiler mechanisms aimed at mitigating program vulnerabilities including Address Space Layout Randomization, and Data Execution Prevention, d) sandboxing certain applications.

Monitoring for known malware or suspicious behavior is not a bullet-proof solution [15, 31]. Although it is necessary to patch software vulnerabilities, this strategy is not very effective in protecting end-users due to slow adoption of software patches by end-users [38, 41]. Exploit mitigation techniques make it harder for attackers to attack benign programs, but not impossible, especially considering new attack methods such as return-oriented programming [18, 24]. In case of sandboxing, it is only applied to certain applications e.g. applications distributed through Mac App Store. Moreover, once malicious code or data gets past these obstacles, it can damage and steal all user data without restriction.

4.1.2 System Compartmentalization

A practical solution to this problem must not require radical changes to operating systems or existing applications since such a solution would be met with huge resistance from software vendors that have made considerable investments in their software products.

System compartmentalization can be used to mitigate the problem without changing operating system APIs or applications. The idea is to partition the execution environment into multiple containers each having limited access to user data and system resources. Two existing systems that employ this technique are Bromium [12] and Qubes OS [60]. Both

systems use full virtualization based on Xen [49]. These two systems represent two different design points in the design space of system compartmentalization solutions.

4.1.2.1 Qubes

Qubes exposes the management of containers to the user through graphical user interfaces and does not help the user in how they would compartmentalize their activities. However, unlike Bromium, Qubes provides compartmentalization for device drivers in addition to user programs. Qubes uses full virtualization without optimization which normally has high overhead, slow to boot containers, and cannot handle more than a dozen containers.

4.1.2.2 Bromium

Bromium does not expose container management to the user, it automatically isolates user tasks that deal with untrusted data in MicroVMs. A user task can be opening an untrusted PDF document or navigating to a website in a single browser tab. Bromium uses memory throttling and other optimizations to reduce the full virtualization overhead dramatically for a limited set of supported applications. The downside of this method is that it only works for supported applications. Unsupported applications revert back to legacy mode with high overhead and no access to user data when executed in a VM. In order to open files with an unsupported application, the user needs to run that application without compartmentalization. Another downside is that it requires changing some applications such as the browser to achieve fine-grained compartmentalization, i.e. isolating tabs.

4.2 An Alternative Approach

FlexICon is a new approach to system compartmentalization that addresses some of the shortcoming of Bromium and Qubes. FlexICon is designed to achieve the following goals: (a) no modification to applications, (b) high performance, (c) reducing the effects of user mistakes, and (d) security. To achieve these goals FlexICon employs the following design elements:

- Light-weight virtualization based on Linux namespaces to achieve low overhead.
- A declarative language for defining containers by specifying resource restrictions for each container and rules for assigning processes to containers.
- A trusted file dialog for applications running in containers which allows the user to pick any file and if necessary open the selected file in another container.
- A browser extension to provide seamless transition between browser instances running in different containers.

Using light-weight virtualization affords FlexICon low performance overhead and small memory footprint without the need to modify or exclude applications. The container specification language allows FlexICon to enforce resource (files and network end-points) restrictions for applications running in containers and automatically decide which container to use for each user command (opening a document or navigating a website). The enforcement of restrictions and automation of container assignments relieve the user of making these decisions at runtime and reduce the effect of user mistakes on security. The trusted file dialog has two main benefits: (a) provide seamless access to all user files in the dialog, (b) provide a mechanism for transferring data between containers when needed by direct user designation. Finally, the browser extension enables the user to click on any link in any browser instance and have the system open the link in the proper browser instance.

4.2.1 Overview of FlexICon

FlexICon containers are restricted execution environments for user applications. There are several configuration options for each container, including: (a) folders that are directly accessible in the container, (b) network end-points (fully qualified domain names) that are accessible in the container, (c) applications that can execute in the container. Note that a given application can be restricted to be executed in a given container or be allowed to have different instances execute in different containers. If multiple instance of the applications execute in different containers, they do not communicate with each other. By default, every container has a dedicated folder in the user's home directory within which applications running in that container have regular access to files and folders. It is also possible to grant access to other folders in the user's home directory if desired. Containers have read-only access to system files and no access to the user's home directory or other containers by default. It is possible to disable network access for a container, or allow restricted access by black-listing / white-listing domain names.

It is also possible to specify rules for assigning processes to containers. When the user wants to launch an application, either directly e.g. by clicking on an application launcher or indirectly e.g. by opening a document, FlexICon consults a list of rules to decide which container to use. These rules can be specified based on the following inputs: (1) the application executable path, (2) any files passed as arguments to be opened, and (3) any URLs passed as arguments to be navigated. FlexICon parses each user command to find the above information and then checks the rules to decide which container should be used to execute the command.

These two main mechanisms (containers and rules) provide great flexibility for isolating application instances, especially when combined with other mechanisms in FlexICon.

Before using the system, one has to specify containers and rules, and FlexICon provides a language for this purpose. The container and rule definitions can be prepared by the user or someone with technical knowledge when installing FlexICon. The remainder of this section explores various types of containers that can be created in FlexICon and introduces other mechanisms available in FlexICon.

4.2.1.1 Web-based Containers

FlexICon containers can be used to compartmentalize web access to gain benefits similar to those gained by Auto-FBI. To compartmentalize web access, one can partition the set of all websites into disjoint classes and make sure that each class of websites is accessed by a separate browser instance.

FlexICon can enforce this policy by: (a) defining one container per class of websites, (b) restricting the network access of such containers so that only domain names of websites allowed in the respective class are accessible in the container, (c) adding rules that would launch the browser in the respective container when a URL belonging to a certain class is passed as argument. For example, if the user is particularly concerned about protecting their banking data, he/she could have two classes, Banking for online banking and General for everything else.

However, without any extra mechanisms, the above model would require the user to specify a URL when launching the browser which would be cumbersome. FlexICon provides a browser extension that can solve this issue. The browser extension reports all user-requested URLs to FlexICon so that if the current browser instance cannot access the requested URL, FlexICon would open the URL in the proper container. Additionally, a rule is needed to execute the browser in the General container when launched directly. These

mechanisms together allow the user to navigate to different classes of websites in separate containers without much hassle.

Additional rules can be added so that when the user tries to open documents stored in a web-based container, the documents are opened in that same container. For example, a PDF file that was downloaded in the General container, will be opened in the General container. These rules would ensure that any potential damage from such documents does not spread to other containers.

Note that in order to prevent CSRF attacks from a malicious site in one container targeting a site in another container, FlexICon removes the path and query parts of the URL before opening it in the target container. Otherwise, the protection gained by separating websites in multiple containers would be lost since the attacker can simply trick the user into clicking on a CSRF link and the system would open the link in the target container.

4.2.1.2 The User Container

When the user launches an application directly, FlexICon has no information about data that will be processed by that application. A container with no access to network, dubbed the *User* container, can be used to launch certain applications in this case. Applications that are typically used to generate content such as text editors are good candidates for launching in the User container when launched directly. This would cause all user-generated contents to be stored in a single container.

Note that it would be a bad idea to have multiple launchers for every application so that the user chooses the container in which the application is to be launched, because it forces the user to make a security decision prematurely. For instance, when the user wants

to open a text editor to type something, the user would need to decide which container is most suited for the intended content before even starting.

To enable users to save data generated in the User container in other containers, FlexICon provides a mechanism that allows applications running in a container to save a file in another container through a trusted file dialog. To enable this mechanism for a container, the container definition should explicitly list which containers can be chosen by the user to save a file through the trusted file dialog, and if no target container is listed in the container definition, then this mechanism will be disabled for that container.

4.2.1.3 The Junk Container

To run unknown applications without worrying about such applications stealing user data or compromising the integrity of the user data, such applications can always be executed in the Junk container. This container is defined with minimal access to file system and no network access (to prevent malicious software like distributed denial of service botnets).

Known but untrusted applications could be assigned to separate containers if they store data that is valuable to the user.

4.2.1.4 The Email Container

Since email client applications deal with data from different sources that might not coexist in a single container, it is best to run email clients in a dedicated container. There are mechanisms in FlexICon designed for email use cases that are discussed next.

A common avenue for attackers to gain control of personal devices is to send malicious attachments through email. The email container can be configured to open attachments

in *disposable* containers. A disposable container is a single-use container that is deleted after being used and only runs a single application instance. By opening email attachments in disposable containers, the possibility of compromising the email container as well as other containers is eliminated. To be able to store trusted attachments in other containers, the trusted file dialog (the same mechanism that is used by the User container discussed earlier) can be used.

To enable the user to attach files stored in other containers to an email, the trusted file dialog would make the files selected by the user available to the email client for a limited amount of time. This mechanism is specially designed for the email use case and is not the default behavior of the trusted file dialog. Similar to the save mechanism, a list of source containers must be explicitly specified in the container definition.

4.2.2 Policy

FlexICon policies enforce restrictions on the flow of data in the system. In order to describe these restrictions some terminology is needed. A data flow is *explicit* if the data flow requires explicit user interaction e.g. by selecting a file in a file dialog, otherwise it is considered an *implicit* data flow. In a personal computer running an state of the art operating system without containers, a program started by the user can access all data owned by that user and can leak such data without user's knowledge to any remote host over the network. In FlexICon, there are some implicit data flows possible, e.g. a program running in a container can access data stored in that container without user's knowledge and leak it to domains accessible in that container. It is also possible to have explicit data flows between containers through the file dialog mechanism. Reasoning about possible data flows allowed

by a policy can be done by building a flow graph. The nodes in the graph represent the following types of entities:

- Data: a folder containing user data
- Processes: processes running in a container
- Origins: a set of web origins (domain names)
- File dialog: a trusted file dialog
- URL trimmer: removes the path and query part of a URL before opening it in the proper container

Each container definition can be directly mapped to a set of nodes and edges. For example, a banking container would have its own data node representing the folder dedicated to the banking container, a process node representing all processes running in the banking container, an origins node representing all banking websites. If the container is configured to be able to read or write data from/to other containers, file dialog node(s) should be created as well. Figure 5 shows how to connect these nodes for one container.

If a rule's target is not a disposable container, then no new data flow is created by the rule. A rule with a disposable container target would induce a flow edge from the data node to the processes node of the disposable container.

Finally, the processes nodes for all network-enabled containers must be connected to a URL trimmer node and the URL trimmer node must be connected to all origins nodes. These connections represent the URL handling mechanism in FlexICon: a browser extension in each container reports all user-requested URLs to the controller module and the controller module would ensure that the URL is opened in the proper container after removing the path and query parts of the URL. A data flow path that passes through the URL filter node would be a low bandwidth channel, because the protocol and hostname, which

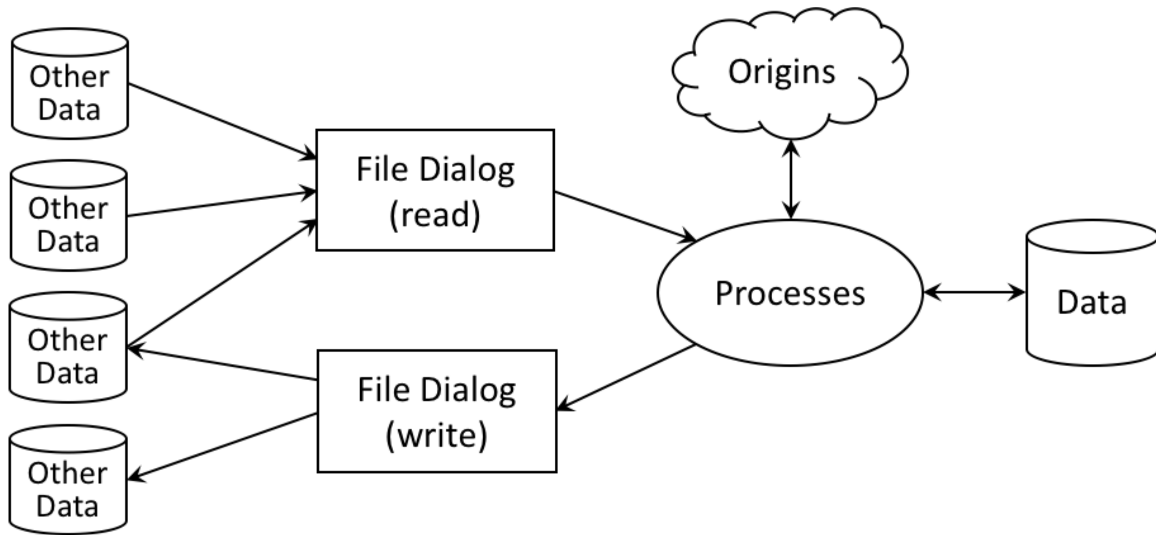


Figure 5. Flow graph nodes and edges for a single container

are preserved by the trimmer, are restricted to HTTP or HTTPS for the protocol and 255 characters for the hostname. If the container is malicious, it can communicate with a host indirectly by making requests for URLs at that host, that would be serviced by a browser instance at another container. Such leakage can be severely restricted by adding checks on the rate of such requests, but the prototype implementation does not enforce rate limiting.

After building such a graph, one can answer data flow queries such as *"can data stored at location X implicitly flow to origin Y?"*. An implicit flow would be represented by a directed path that does not go though any file dialog nodes and an explicit flow is analogous to a directed path that passes through at least 1 file dialog node.

4.3 Threat Model

4.3.1 Scope

The target is a personal user system in which the user is not particularly motivated or inclined to invest effort in security. The user can browse content from a variety of websites. Some of these websites are potentially malicious and some can be reasonably expected to be non-malicious. Malicious websites include both legitimate sites that have been compromised by attackers to disseminate malware (this is the largest source of malware dissemination [58]) as well as those that are launched by attackers who lure users into visiting their websites. Non-malicious websites include sites that could be reasonably expected to be secure, such as sites of banking institutions, or sites for large organizations that can be expected to quickly patch any compromise.

This threat model includes an attacker who can surreptitiously download content and execute it on the user system (drive-by download) if the user visits a site under the attacker's control. This attack subsumes other attacks such as CSRF, XSS, and Clickjacking, so it will be the only attack assumed in this chapter. A drive-by download attack is assumed to be able to take over the container in which the browser is executing.

It is assumed that the operating system mechanisms (including Linux namespaces [10] and TUN/TAP [46]) are secure. The implementation of FlexICon is also assumed to be secure. The point is to argue relative security.

Some applications are trusted if not executing malicious content and some other applications are not trusted.

4.3.2 Security Goals

The main security goal is to protect the integrity of the user data from attackers (ransomware for example [34]) as well as providing some confidentiality. The integrity protection is paramount because compromising content in sensitive containers can compromise the whole container. While confidentiality is important, if one wants to allow the user seamless access to data, allowing content from sensitive containers to be read by other containers is unavoidable. This is needed to support an email application that can aggregate content from multiple containers and cannot be allowed to execute in a sensitive container. So, while absolute confidentiality is not a goal, all breaches of confidentiality must be initiated by the user or be done over a low-bandwidth channel. This limits the rate at which a compromised container can siphon information from other containers. If other measures are in place to clean-boot non-trusted containers, this will be effective in providing confidentiality.

4.4 Design & Implementation

This section presents the design and implementation of FlexICon.

4.4.1 System Design

FlexICon is composed of multiple modules. Figure 6 shows an overview of these modules and their relationships. The following is a high level description of these modules.

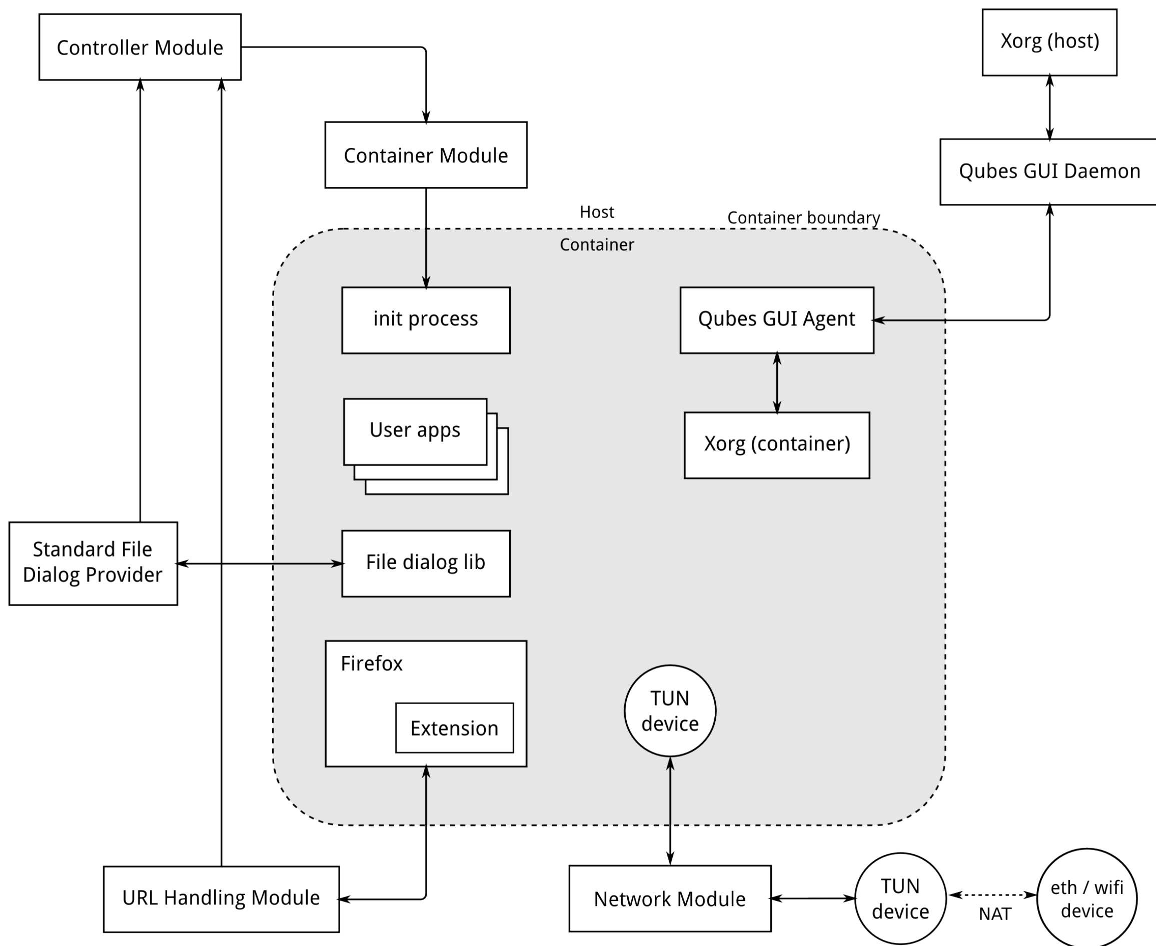


Figure 6. FlexICon's system architecture

4.4.1.1 Controller Module

The controller module is responsible for managing containers at runtime. It reads the policy file to get container definitions and rules. It prepares configuration files for the container and network modules based on container definitions in the policy. The controller is also responsible for parsing user commands and assigning them to containers based on the rules.

4.4.1.2 Container Module

The container module is responsible for creating a single container. It does so based on the configurations provided by the controller module which is in turn based on the container definition in the policy. The container module employs a series of Bash scripts to setup various aspects of the container, including mounting the system files and the container's home directory, setting up network devices and launching the network module, the X window server and the GUI module. After setting up the container environment, the container module waits for commands from the controller module to execute user commands in the container environment.

4.4.1.3 Network Module

The network module is responsible for filtering network traffic of the container. It has an internal white-list of IP addresses and it checks all outgoing IP packets and only allows packets with a white-listed destination IP address. It also monitors DNS messages to update its IP white-list when the domain name in the DNS message is allowed by the policy.

4.4.1.4 Trusted File Dialog

The trusted file dialog runs outside all containers and can present all files to the user for selection. Based on its configuration settings, it can provide the following mechanisms: (a) opening selected files in their proper container, (b) saving data from one container to another, and (c) reading data stored in a different container. Mechanism (a) is the default behavior while the other mechanisms must be explicitly enabled for each container. It is

also possible to limit the possible destinations and sources of data for mechanisms (b) and (c). For instance, it is possible to allow the email container to read data from a subset of containers to avoid leaking sensitive data by mistake.

4.4.1.5 URL Handling Module

The browser extension reports all user requested URLs, i.e. URLs typed in the address bar, or a link / bookmark that was clicked by the user, to the URL handling module. The URL handling module consults with the controller to figure out if the browser instance reporting the URL is allowed to access the reported URL and if not, it would ask the controller to open the URL in the proper container after removing the path and query parts of the URL.

4.4.1.6 GUI Module

The GUI module is adapted from Qubes OS [60]. Every container gets a local X window server and the GUI module creates a bridge between the container's X server and the X server running outside all containers which has access to the graphics hardware. The GUI module ensures that applications running in separate containers cannot spy on each other or inject fake keyboard or mouse events. It also provides secure copy/paste between containers through special keyboard combinations (Ctrl-Shift-C and Ctrl-Shift-V).

4.4.2 System Implementation

The FlexICon prototype is implemented for Ubuntu 14.04 with KDE desktop. This section provides an overview of the implementation and highlights novel technical aspects

of it. The implementation consists of 14360 lines of C/C++ code plus 1430 lines of Bash scripts. Modules adapted from Qubes [60] are not included in these statistics.

To create containers Linux namespaces [10] are used similar to the Linux Containers (LXC) project [48]. A notable difference in FlexICon's implementation is the use of read-only bind mounts to mount system files in each container which allows it to use a single software stack.

When the user launches an application from the launcher menu or clicks on a file to open, the result is a command that includes the application's executable path and zero or more arguments that may or may not be interpreted as files or URLs by the target application. The controller module is responsible for parsing these commands. An interesting challenge is to correctly identify arguments that would be interpreted as files or URLs by the target application so that policy rules can be applied to the user command to figure out which container should be used. FlexICon interprets a given command based on its knowledge of how the target application would parse its options (information that can be typically found in man pages).

The standard file dialog in KDE is amended to show a trusted file dialog running outside of the container and let the user pick any file. The system then decides how to handle the user's selection in accordance with its policy. Normally, if the user chooses to open a file that is already available in the container from which the dialog was launched, it just reports the selected file back to the caller. However, if a file is picked that is not available in the caller's container, the system opens the selected file in the proper container. Other mechanisms of the trusted file dialog, namely the "save file in another container" and "read file from another container", are implemented by creating hard links.

To enable graphics display access in containers, the GUI module from Qubes OS [60]

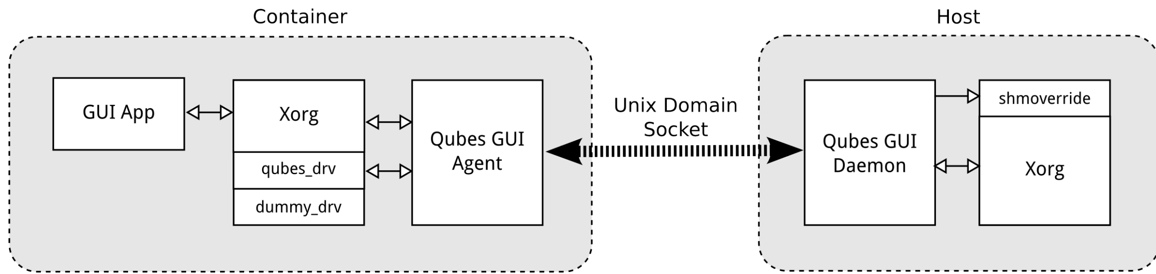


Figure 7. Adapted Qubes GUI module

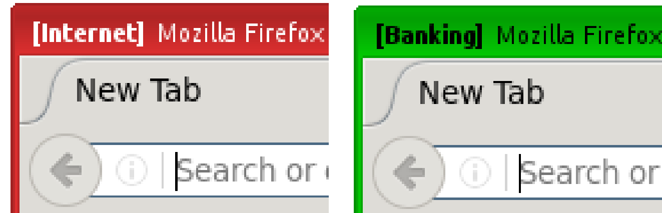


Figure 8. Example window border colors

was adopted. Each container gets its own instance of the X window server with a dummy video driver that is backed by a buffer in main memory instead of access to real graphics hardware. The Qubes GUI Agent running inside the container alongside the X server (Xorg in the diagram) acts as a desktop manager for the container and communicates with the Qubes GUI Daemon running outside the container alongside the main X server that has access to graphics hardware. The image of a window of a program running inside the container is rendered on the actual graphics display through shared memory pages. The Agent and Daemon modules also communicate keyboard and mouse events for applications running in the container. Figure 7 shows details of these modules. To adapt these modules from Qubes OS which uses Xen virtualization, parts of the code responsible for communication between the agent and the daemon were modified to use Unix domain sockets.

The windows belonging to applications running in containers are shown with colored borders and the name of the container is shown in the title-bar. The border color is configurable for each container and can be chosen from 8 predefined colors. A different color

is reserved for windows of processes running outside any container. The border color can be used for the following purposes: a) to convey the level of trust for applications, documents and websites opened in that container, or b) to easily distinguish windows belonging to different containers. Example colored window borders and title-bars are shown in Figure 8. Even windows that do not have a title-bar, for example right-click context menus, are rendered with a thin border of the same color.

To enable networking in containers a pair of TUN [46] virtual network devices are created and a user-space program sits between the TUN devices to control network traffic (see Figure 6). The network filtering module can be configured to allow or disallow traffic based on fully qualified domain names as well as IP addresses. Internally, it has a white-list of IP addresses and it inspects DNS responses received through the tunnel to associate DNS names with IP addresses. The details of container networking is illustrated in the next section with an example.

4.4.2.1 Container Networking

In the following discussion it is assumed that there is a Banking container that is allowed to access www.bank.com but not www.example.com. The interactions of a browser running in the Banking container with the network module is illustrated in Figures 9, 10 and 11.

The network module instance serving the Banking container is set up in the following way: `TUN1` has IP address `169.254.1.1`, `TUN0` has IP address `169.254.1.2`, `dnsmasq` is listening for DNS requests on `TUN0` and finally the nameserver is set to `169.254.1.2` in the container's `/etc/resolv.conf` file. The network module adds the IP address of `TUN0` to its internal white-list upon starting to allow DNS requests to go through.

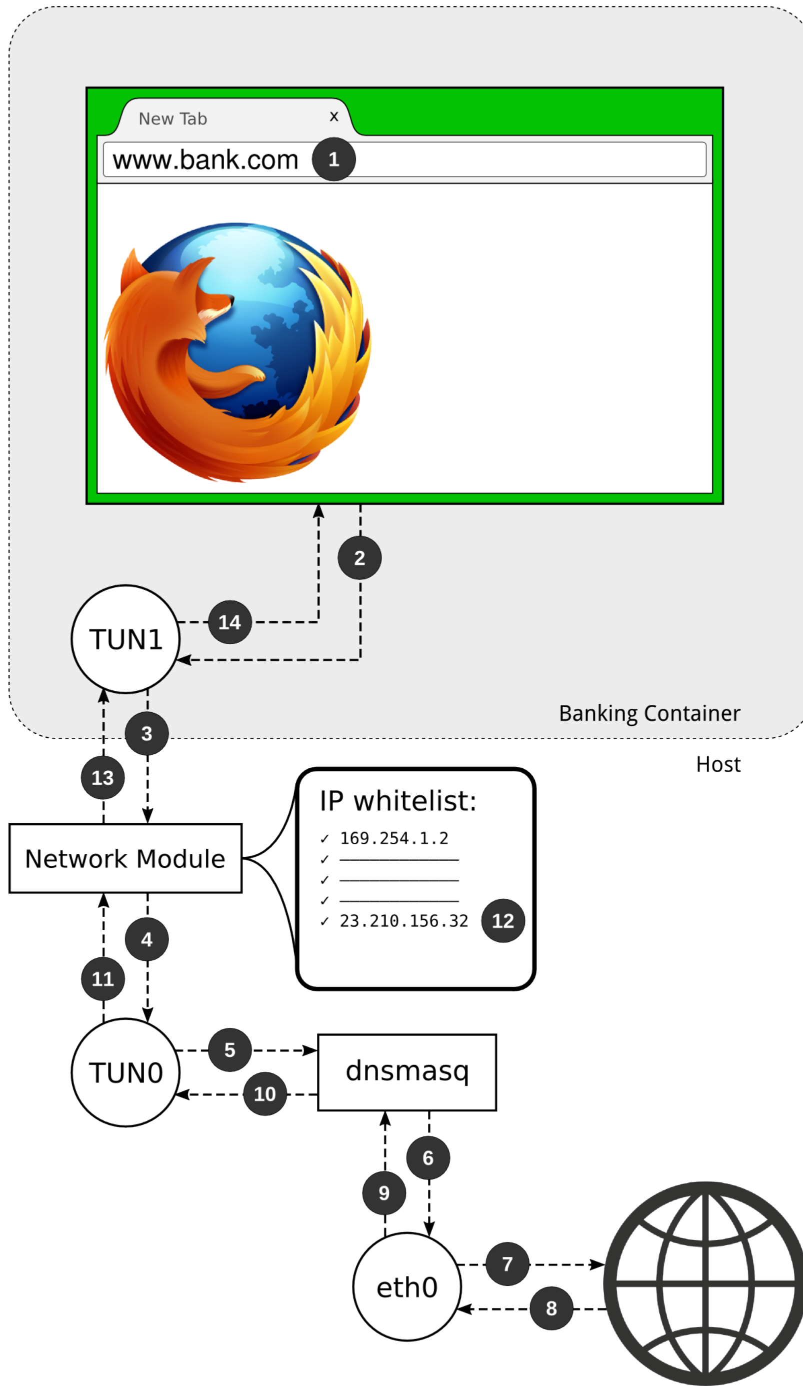


Figure 9. Resolving a DNS name

Accessing www.bank.com The numbers in parentheses refer to steps depicted in Figure 9:

(1) The user instructs the browser to navigate to www.bank.com.

(2) The browser sends a DNS request to find the IP address for www.bank.com, the DNS request has the following destination IP address: `169.254.1.2` due to the `nameserver` setting.

(3) The network module receives the DNS request packet, it checks the destination IP address and allows the packet to pass through since the IP address `169.254.1.2` is listed in its internal white-list.

(4) The DNS request packet is sent out to `TUN0`.

(5) The DNS client (dnsmasq) listening for DNS requests on `TUN0` receives the packet.

(6,7,8,9) The DNS client resolves www.bank.com by accessing the Internet through the real network device eth0.

(10) The DNS client generates a DNS response mapping www.bank.com to IP address `23.210.156.32` and sends it back to the browser through `TUN0`.

(11) The network module receives the DNS response.

(12) The network module examines the DNS response and consults its configuration file to see if www.bank.com is allowed. Since www.bank.com is allowed, it adds the corresponding IP address `23.210.156.32` to its internal white-list.

(13) The network module lets the DNS response to go through.

(14) The browser receives the DNS response and figures out the IP address for www.bank.com.

At this point the browser can initiate an HTTP connection to the bank website by using the IP address acquired in the previous steps. The numbers in parentheses in the rest of this discussion refer to steps depicted in Figure 10.

(15) The browser starts accessing www.bank.com by sending an IP packet to `23.210.156.32`.
(16) The network module receives the packet.
(17) The network module checks the destination IP address, `23.210.156.32`, against its internal white-list and since it is listed there, it allows the packet to go through.
(18) The kernel routes the IP packet using NAT to the real network device eth0.
(19) The packet is sent over the Internet.
(20) The web server for www.bank.com sends a response which is received by the real network device.
(21) The kernel routes the packet to `TUN0` following NAT rules.
(22) The response packet is received by the network module.
(23) The network module allows the packet to go through.
(24) The browser receives the web server response.

Accessing www.example.com The first steps for accessing www.example.com is the same as those described earlier (depicted in Figure 9) with one exception: step (12) will be executed differently. Since www.example.com is not allowed according to the network module's configuration, the IP address will not be added to the white-list. The browser acquires the IP address for www.example.com i.e. `93.184.216.34`, but that IP address is not added to the internal white-list of the networking module. The numbers in parentheses refer to steps depicted in Figure 11:

(1) The browser starts accessing www.example.com by sending an IP packet to `93.184.216.34`.
(2) The network module receives the packet.

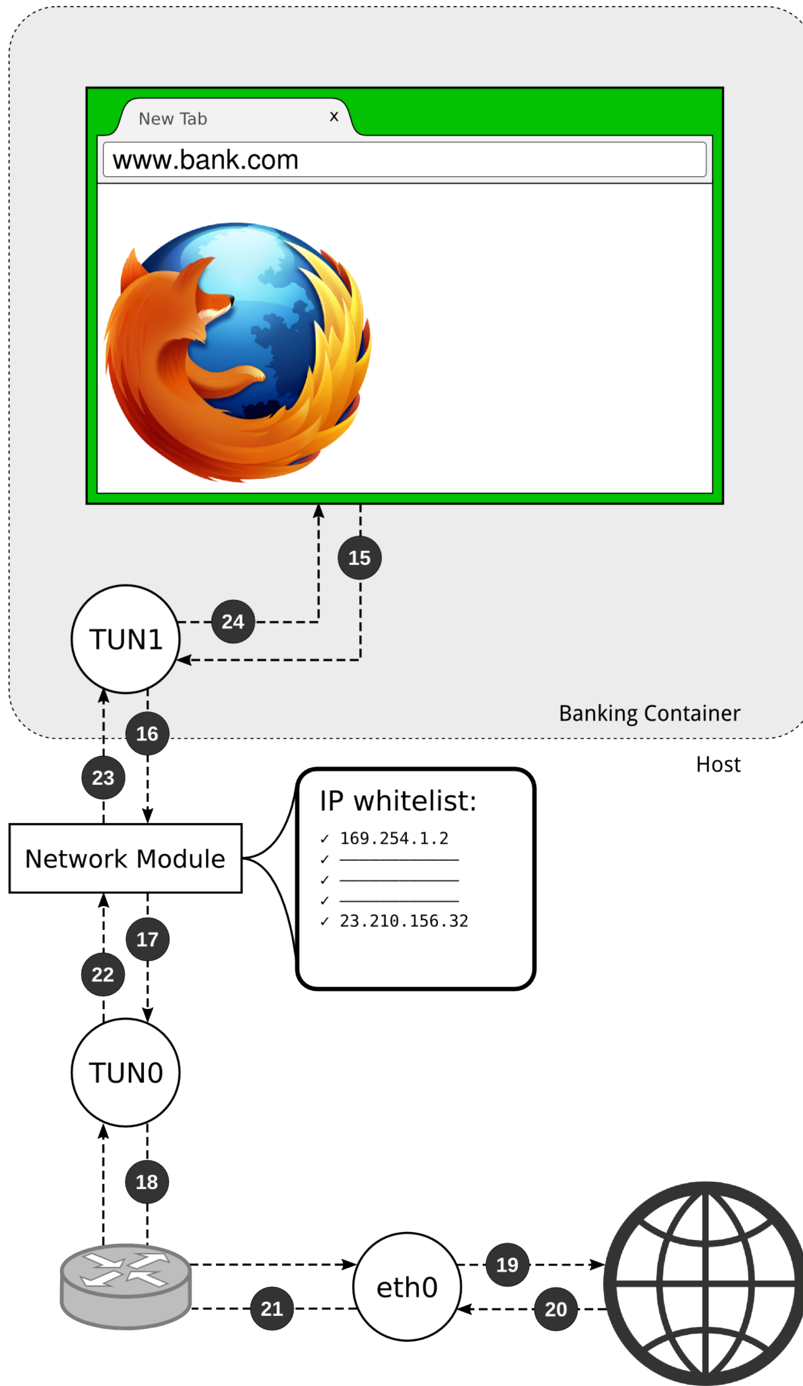


Figure 10. Successful IP connection

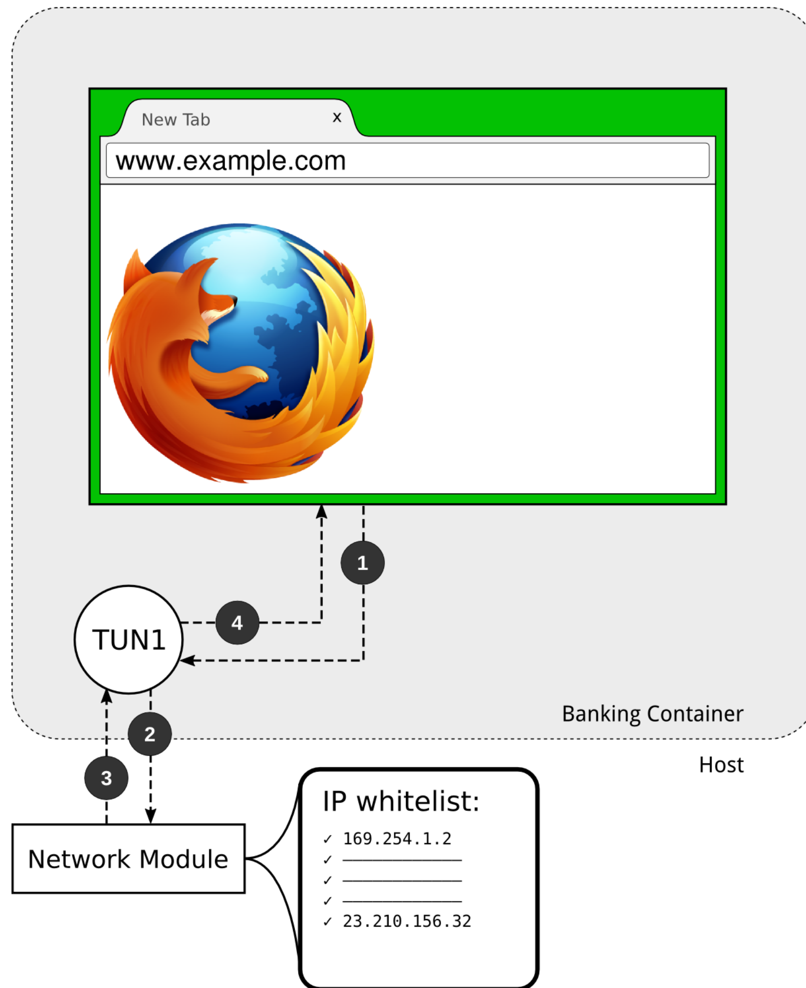


Figure 11. Connecting to an unlisted IP address

(3) The network module checks the destination IP address `93.184.216.34` against its internal white-list and since it is not listed there, it drops the packet. It then generates an ICMP Destination Unreachable response for the browser.

(4) The browser receives the Destination Unreachable response.

```
# Container definition
container "Sensitive" {
    color_index: 4
    gui: true
    system_files: normal
    home_extras: []
    save_as_other_containers: []
    read_from_other_containers: []
    network_access: true
    domains {
        + "[*.]bank1.com",
        + "[*.]bank2.com",
        - "*"
    }
    usr_bin { + "*" }
}
```

Figure 12. Sample FlexICon container definition

4.4.2.2 Configuration Syntax

The configuration file in FlexICon is composed of two main parts: (1) container specifications, and (2) rules. The first part, defines all containers in the system. It specifies what resources are available in each container. Figure 12 shows an example container specification. Note that container definitions start with the keyword `container` while disposable containers are defined with the `disposable` keyword instead. The configuration options for each container are as follows:

- Name: a string that uniquely identifies the container and is used to refer to that container in rules.
- Color index: window border color specified as a number between 1 to 8.

- GUI: does the container need GUI capability.
- System files: can be used to further restrict access to system files.
- Home extras: other folders in the user's home directory that must be accessible in the container.
- Save as other containers: list of container names that can be selected by the user in the trusted file dialog to save data originating from this container.
- Read from other containers: list of container names that can be selected by the user in the trusted file dialog to read data from.
- Network access: whether or not the container should be allowed to access network.
- Domains: a black/white list of domain names that are allowed in the container.
- User binaries: black/white list of programs in `/usr/bin` available in the container.

The second part of the configuration file (list of rules) is used to determine which container should be used to run a given command. Figure 13 shows two example rules related to the previous example container. Given a user command, if all conditions in the body of a rule are satisfied, the command is executed in the target container. Rules that are listed first in the policy have higher priority. Each rule has the following parts:

- A black/white list of applications that are accepted by the rule.
- An optional black/white list of files that are matched against the files passed as arguments to the application.
- An optional black/white list of URLs that are matched against the URLs passed as arguments to the application.
- A target container that will be used to launch the application if the rule matches the given command.

```
# Rule 1: open sensitive files in Sensitive
if {
    application {
        + "/usr/bin/firefox",
        + "/usr/bin/okular",
        + "/usr/bin/kate",
        - "*"
    }
    file {
        + "/Sensitive/*",
        - "*"
    }
} then "Sensitive"

# Rule 2: open bank URLs in Sensitive
if {
    application {
        + "/usr/bin/firefox",
        - "*"
    }
    url {
        + "^://[^.]bank1.com[/*]",
        + "^://[^.]bank2.com[/*]",
        - "*"
    }
} then "Sensitive"
```

Figure 13. Sample FlexICon rule definitions

```
# Define bank URLs as a list of strings
define BANK_URLS [
  "^://[^.]bank1.com[/*]",
  "^://[^.]bank2.com[/*]"
]
# Define a white-list for browsers
define BROWSER { + "/usr/bin/firefox", - "*" }
# Rules using previous definitions
if {
  application BROWSER
  url { + BANK_URLS, - "*" }
} then "Sensitive"

if {
  application BROWSER
  url { - BANK_URLS, + "*" }
} then "General"
```

Figure 14. Avoiding repetition with definitions

Wildcard expressions can be used in specifying black/white lists. The following are special characters that can be used in wildcard expressions:

- '*': matches a sequence of zero or more characters
- '^': matches a sequence of zero or more characters excluding slash (/)
- '[]': can be used to specify an optional segment
- '\': can be used to designate the literal value of any of the special characters

The configuration syntax also allows for defining strings, lists of strings and black/white lists using the `define` keyword to avoid repetition. Figure 14 shows an example of using such definitions to avoid repetition. A full example configuration is listed in Appendix A.

4.5 Security Evaluation

In security, improvement depends on the particular policy specified. However, the following can be stated about FlexICon in general:

- **Restricted Access:** Malicious code or data running in a container can only compromise user data that is available in that container and that can be specified to be as little or as much as the user needs.
- **Browser Security:** It is possible to access different websites in separate containers, to eliminate the possibility of cross-origin attacks. This is also useful for protecting the browser state (cookies, saved passwords, history). Each container will store browser state related to websites assigned to that container only.
- **The Human Error Factor:** An important aspect of FlexICon is the ability to specify a security policy in a rule-based language and mechanisms that allow seamless navigation in the user interface. The automatic enforcement of the policy means that the user is less involved in making security decisions, which, according to HCI research [22, 40], would reduce human error and improve security. In comparison, Qubes does not reduce the burden of decision making on the user which could have an adverse effect on security.

4.5.1 Attack Surface & the TCB

In order for a container to successfully "contain" a process, the process must not be able to bypass the isolation mechanism. The attack surface that the system presents can give an indication of the security of FlexICon relative to Bromium and Qubes if one does not consider the benefits of automatic enforcement of policy. A quantitative analysis of the

attack surface like the one described in [52] is beyond the scope of this dissertation, but a qualitative analysis is provided next.

FlexICon's containers use Linux namespaces as well as other Linux system calls to provide isolation. A single Linux kernel is shared by all processes running in all containers which affords FlexICon a very low performance overhead, however, since Linux is a monolithic OS, this also means that a vulnerability in the kernel could be used by a malicious process to change the internal state of the kernel in order to bypass the isolation. In comparison, Qubes and Bromium provide isolation using full virtualization (both based on the Xen hypervisor). The attack surface of the isolation mechanism in a Xen-based system includes the Xen hypervisor and the software running in the control domain (or dom0 using Xen terminology). The control domain is responsible for emulating hardware for other domains and launching new VMs. Both Bromium and Qubes run a full software stack in the control domain although Qubes further disaggregates this domain by running network and USB drivers in separate domains. Comparing the attack surface of FlexICon to that of Qubes or Bromium is not easy; although the size of TCB seems comparable, the complexity and number of interfaces exposed are not the same. At a high level, interfaces exposed by Xen and its control domain are simpler and fewer than those exposed by the Linux kernel. Therefore, one can argue that Bromium and Qubes provide stronger security guarantees compared to FlexICon from an attack surface perspective.

Other FlexICon components that might be targeted by malicious processes in order to bypass the isolation should also be included in this analysis. Components in Figure 6 that are outside the boundary of the container run directly on the host in order to provide services for the container. Some of these components accept inputs from inside the container that could be manipulated by a malicious process. These components are: Qubes GUI daemon, network module, URL handling module and the standard file dialog provider. A

vulnerability in these components could enable a malicious process to escape the isolation. The total size of these components is about 8.8 KLOC.

All 3 systems are susceptible to side channel attacks, but since the kernel is shared among containers in FlexICon, unlike Qubes and Bromium where each VM runs on a separate kernel, FlexICon could be more susceptible to side channel attacks.

4.5.2 Network Attacks

This section discusses different ways that FlexICon's network module could be subverted in a way that Internet domains that are not allowed in a container according to the policy become accessible in that container.

4.5.2.1 Subverting the Network Filtering through DNS

FlexICon's network filtering module uses DNS responses received from the Internet to associate IP addresses with fully qualified domain names. It is assumed that DNS responses are trustworthy. An active network attacker controlling the user's access point to the Internet can subvert the network filtering mechanism by changing DNS responses. This vulnerability stems from the design of DNS which does not provide any authentication mechanism in its basic form [4]. A solution to this problem would be to adopt DNSSEC to detect forged DNS messages. This problem is not specific to FlexICon and it is orthogonal to the problem addressed in this chapter.

An attacker that controls a name server can respond dishonestly to DNS requests for domain names under its control in order to circumvent the network filtering mechanism. Since credentials such as session ids and other cookies belonging to the target website are

not transmitted in this scenario, there is little concern about the outcome of such an attack in general. However, if it is possible for the attacker to associate a domain name with a private IP address, the attacker could gain unauthorized access to unprotected local web consoles such as the WiFi router settings [37]. This can be easily addressed in the network filtering module by preventing private IP addresses from being added to the internal white-list when processing DNS responses.

4.5.2.2 Subverting the Network Filtering through Proxy

Another possibility for accessing websites that are not allowed in a container is to route the requests through a proxy server. If a container is compromised by malicious code, it would be possible for the attacker to change the browser's network settings in that container and set a proxy. In this scenario the browser running in the compromised container might be able to access any website. Note that this attack does not provide access to other containers. Also, a savvy user who notices that the container name shown in the title bar does not match the container that a website is supposed to be rendered in would not enter sensitive credentials on the website. It should also be noted here that malicious code running in a container cannot change the border color and name of the container shown in the title bar since those are controlled by code running outside the container and also full screen windows are not allowed. Nevertheless, usable security research [2] shows that users tend to ignore such passive visual cues and this should not be relied upon as a way to counter this threat. To eliminate the possibility of this attack, other mechanisms for preventing code running in a container from changing browser proxy settings are needed.

4.6 Limitations

The network module in FlexICon relies on DNS to filter container's network traffic. In general it is possible that different domain names are mapped to the same IP addresses, e.g. Google Search and Gmail. Therefore, it is not always possible to isolate two websites in separate containers since allowing one could mean allowing both.

If a container is compromised, the mechanism that allows the browser extension to report URLs to the controller could be abused by malicious code to open large number of browser tabs which can be a nuisance. One could limit the rate of this reporting and tie it to user input activity and use it to detect rogue containers. However, such a solution was not implemented in FlexICon.

FlexICon does not currently support sound in containers. This can be solved for example by running PulseAudio in each container and passing the sound data back to the host through a Unix domain socket similar to what Qubes has done for sound support.

The controller module in FlexICon parses user commands to identify files and URLs passed as arguments. The current implementation supports a limited number of applications.

Chapter 5

COMPARING SYSTEM COMPARTMENTALIZATION SOLUTIONS

This chapter provides a detailed performance evaluation of Qubes, Bromium and FlexICon. The performance evaluations are designed to answer the following questions about these three compartmentalization solutions: (a) how many containers can be created without using swap memory, (b) how long does it take to prepare a container and launch an application in it, (c) disk access overhead, and (d) network access overhead. A qualitative analysis of GUI performance for FlexICon and Qubes is also presented at the end.

5.1 Test Setup

The following software setups were used for performance evaluations:

- **Qubes:** Qubes OS R3.2
- **Bromium:** beta version of Bromium for home users (vSentry 4.0.2.1276) running on Windows 10 Enterprise with policy controlled by Bromium
- **FlexICon:** installed on Ubuntu 14.04.5 with KDE desktop

All tests were performed on the same machine with identical hard drives. The test machine (Dell OptiPlex 990) was equipped with 8 GB of memory and an Intel Core i5-2400 CPU with 4 cores. The hard drives were Western Digital WD5000AAKX (16 MB cache, 7200 RPM). The term *native Linux* refers to Ubuntu 14.04.5 and the term *native Windows* refers to Windows 10 Enterprise edition. The term *container* is used to refer to FlexICon containers as well as Qubes AppVMs and Bromium MicroVMs. The error bars in diagrams show standard deviation.

5.2 Maximum Number of Containers

To find out how many containers can execute simultaneously, new instances of various applications were launched in separate containers until the system ran out of memory. The following applications were tested: the terminal emulator (xterm), the PDF viewer (evince for Qubes, okular for FlexICon and Adobe Reader for Bromium) displaying a 13 page academic paper, the Firefox browser displaying its default start page (for FlexICon and Qubes) and Internet Explorer displaying an empty page (for Bromium), and a simple hello world program written in Go. Of course, in real use there will be a mixture of different applications running simultaneously and the memory usage of applications can increase with user interaction, but this experiment can provide a rough idea of how the system scales.

In Qubes, the system refuses to spawn a new container if there is not enough main memory left. For FlexICon, the amount of swap space used by the system was monitored and the test was stopped once swap usage was greater than zero. For Bromium, the Windows page file was set to the minimum possible value of 16 MB and test was stopped once the system produced error messages regarding memory. The maximum number of containers that could be spawned on the test machine are listed in Table 3. It should be noted that there are 3 default VMs in Qubes that are needed for the system to function properly: dom0, sys-net and sys-firewall. The numbers in the table do not include these 3 VMs.

The results show that with 8 GB of memory, Qubes cannot exceed 16 AppVMs, while FlexICon can achieve much higher number of instances. For Bromium, when Internet Explorer is set to open websites in new tabs, a large number of instances can be created which is the result of the careful engineering that Bromium has done to control the memory usage of the browser. However, interestingly, only 16 instances of the hello world program could be created in Bromium. This is perhaps due to the fact that when encountering unknown

Table 3. Maximum number of containers

Application	FlexICon	Qubes	Bromium
Terminal	164	16	-
Hello world	185	16	16
PDF viewer	83	15	153
Firefox	34	11	-
IE (tab)	-	-	491
IE (window)	-	-	64

applications, Bromium reverts back to what they call legacy VMs which are essentially normal Xen domains. In case of known and supported applications such as Internet Explorer and Adobe PDF viewer, Bromium manages to create many more instances by using memory throttling and other techniques. This strategy requires making changes to applications and would only work for supported applications.

5.2.1 FlexICon's Memory Usage

To measure the memory overhead of FlexICon containers, the following two tests were performed while monitoring the system memory usage as reported by `free`. In the first test, 30 containers running the terminal emulator were spawned. In the second test, 30 containers running the Firefox browser were spawned. The memory usage of the container includes the memory needed by various FlexICon modules as well as the memory used by Xorg and various KDE processes (e.g. `kdeinit4`) running inside the container, but excludes the memory used by the application launched inside the container (e.g. Firefox). The memory usage of the container can vary slightly depending on the application(s) running in the container. This is due to auxiliary system processes that are needed by each

specific application. On the test machine, the memory usage of each container ranged from 32.8 MB to 36.5 MB.

5.3 Container Load Delay

The load delay of containers was measured for FlexICon and Qubes. Note that Bromium does not provide a means to control the loading of MicroVMs. To measure the load delay, the terminal emulator was configured to run a script upon its start that would record in a text file the time it was executed. To calculate the container load delay, the timestamp recorded by the script was subtracted from the timestamp of issuing the command to start the terminal. These measurements were validated using multiple manual wall clock measurements. The container load delay was measured in two scenarios: when loading containers one at a time and when loading concurrently.

5.3.1 Loading Containers One at a Time

To measure container load delay, 14 Qubes AppVMs and 32 FlexICon containers were spawned one at a time. The following external CPU loads were applied: from 0 to 100% load in increments of 25%. In each case the tests were repeated 10 times. The external CPU load was generated using a simple infinite busy loop engaging 0 to 4 CPU cores. Figure 15 shows the average load delay of containers under these scenarios. On average it takes 0.8 to 1.8 seconds to boot a FlexICon container and 9.5 to 15.6 seconds to boot a Qubes AppVM. FlexICon containers load more than 10 times faster than Qubes AppVMs on average.

The reported container load delays include the time it takes to launch the terminal.

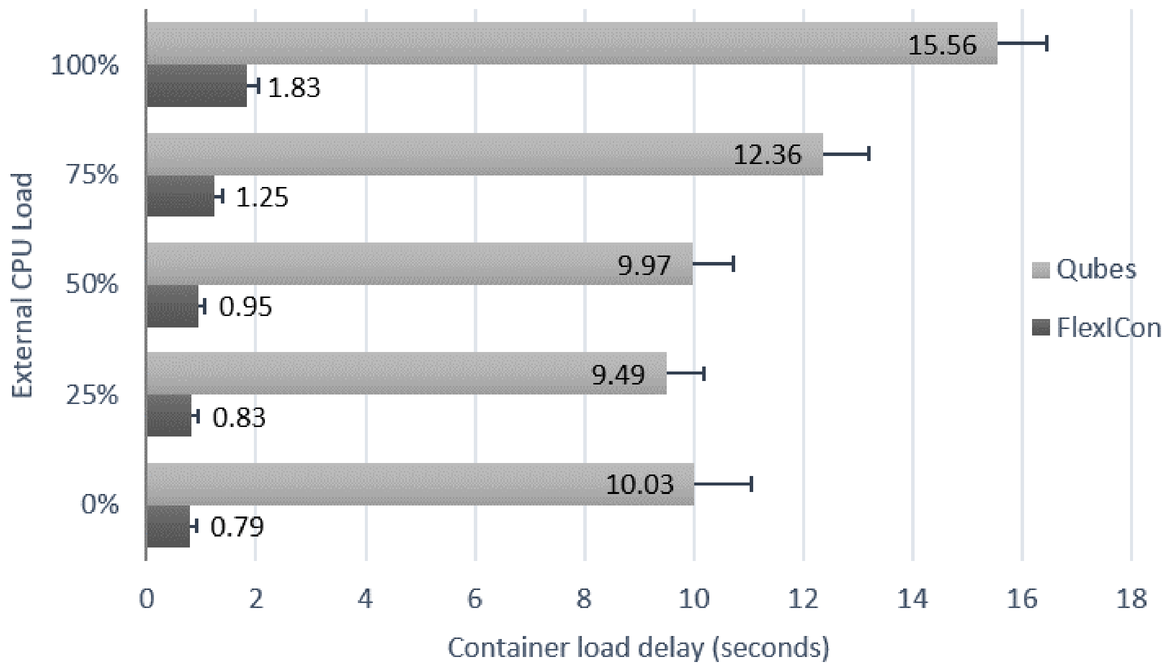


Figure 15. Loading containers one at a time

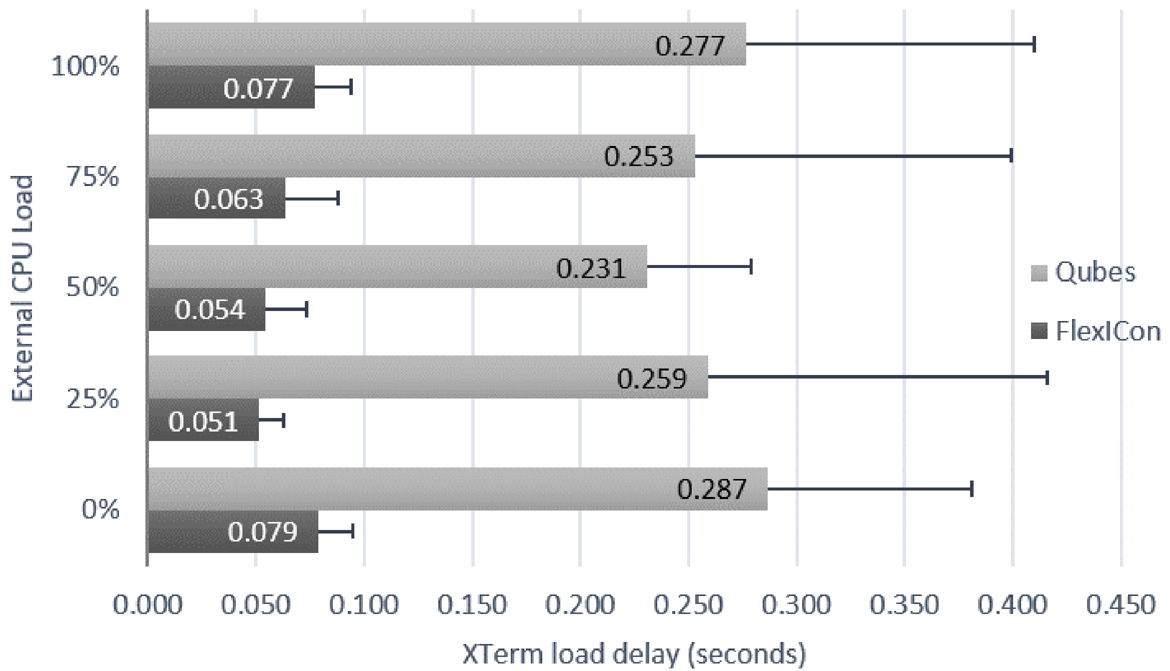


Figure 16. Start delay of xterm in a pre-loaded container

Therefore, the time it takes to launch the terminal in a pre-loaded container was also measured. The results are shown in Figure 16. On average it takes 0.05 to 0.08 seconds to launch xterm in a FlexICon container and 0.23 to 0.29 seconds in a Qubes AppVM. Note that although the same program was used in both platforms, the start delays are very different. This is probably due to better cache utilization in FlexICon[6].

5.3.2 Loading Containers Concurrently

Up to 32 FlexICon containers and 12 Qubes AppVMs were spawned concurrently. Figure 17 shows the average load delay of containers for this experiment. For comparison, it takes 9.4 seconds to launch 32 FlexICon containers, and 10.3 seconds to launch just 2 Qubes AppVMs. Launching 12 containers concurrently takes 3.6 seconds for FlexICon, and 40.5 seconds for Qubes.

5.3.3 Browser Launch Delay

The time it takes to launch the browser was measured on native Linux, a pre-loaded FlexICon container, a pre-loaded Qubes AppVM and a Bromium MicroVM. Firefox was used on native Linux, FlexICon and Qubes. Internet Explorer was used on Bromium. Two scenarios were tested in each case: cold and warm cache (referring to OS disk cache). The cold cache numbers are relevant to a freshly booted system when the cache is not warmed yet, i.e. the first time the user launches the browser after booting the machine. After the first launch, the cache is warmed and subsequent launches will be much faster. To simulate

[6]Linux caches recently accessed files in memory. When starting a container in FlexICon, the system accesses much smaller files compared to Qubes AppVMs, which leads to better cache utilization.

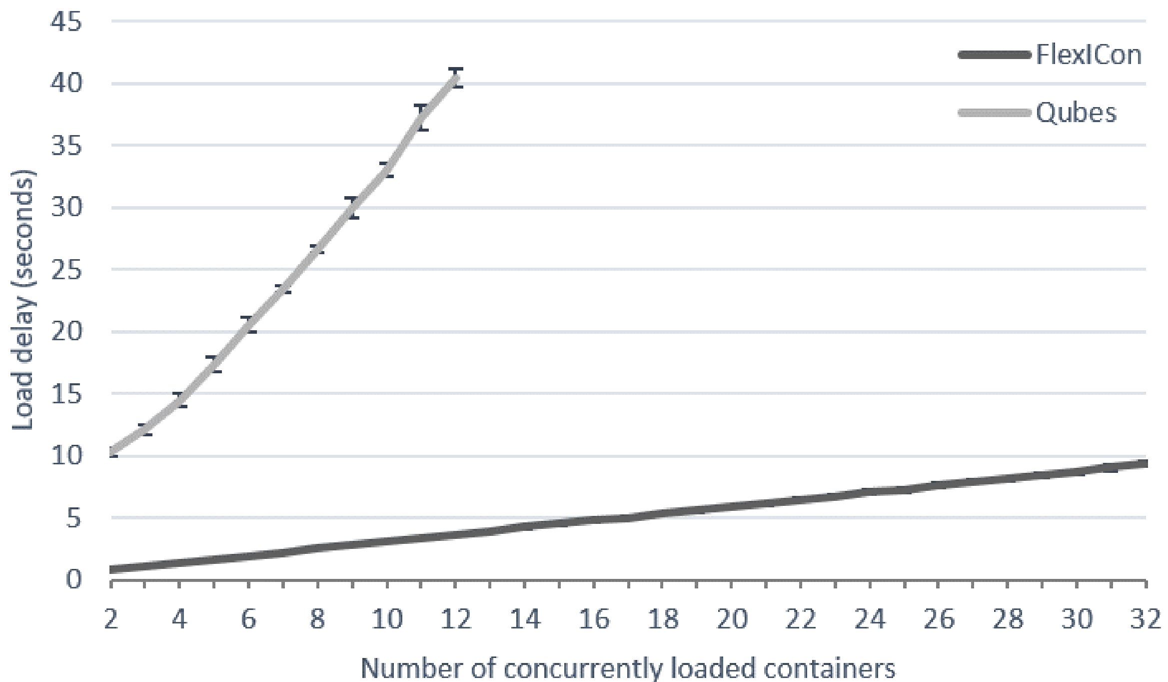


Figure 17. Loading containers concurrently

cold cache on Windows (for Bromium), the Superfetch service was turned off and a simple program was used to allocate huge amount of RAM which forces Windows to drop cached files. On Linux platforms (for native Linux, FlexICon and Qubes), the following command was used to drop caches:

```
$ echo 3 > /proc/sys/vm/drop_caches
```

The results are summarized in Figure 18. The browser launch delay is slightly better for FlexICon compared to native Linux, which is interesting. Overall, considering variations in browser types and versions among the studied platforms, the numbers are similar.

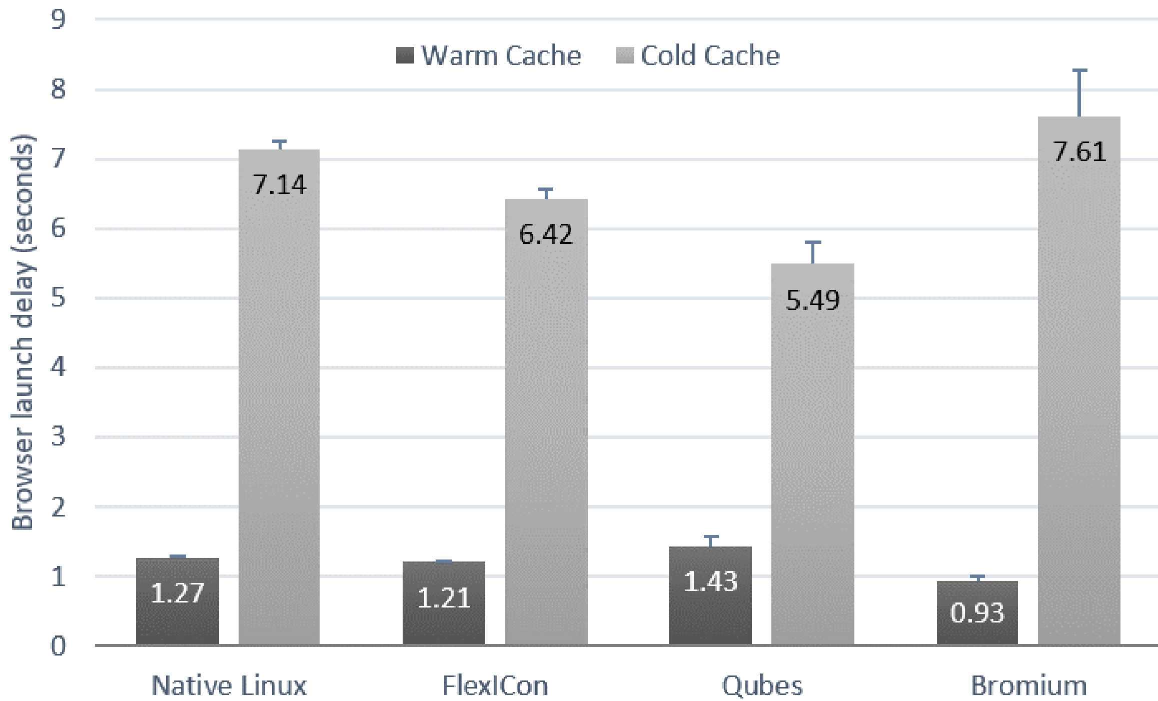


Figure 18. Browser start delay

5.4 Disk Access Overhead

The disk read/write overhead for FlexICon, Qubes and Bromium was measured using two programs written in Go (which can be compiled and executed without change on all platforms). The programs performed the same operations (semantically) on all platforms, specifically in regards to the `Sync()` operation[7] which ensures the data is written to disk before returning. The Go garbage collector was disabled and static buffers were used for read and write. To calculate the overhead, the following comparisons were performed:

- For FlexICon: native Linux vs. a FlexICon container
- For Qubes: dom0 vs. an AppVM
- For Bromium: native Windows vs. a MicroVM

[7] See `os.File` type at https://golang.org/pkg/os/.

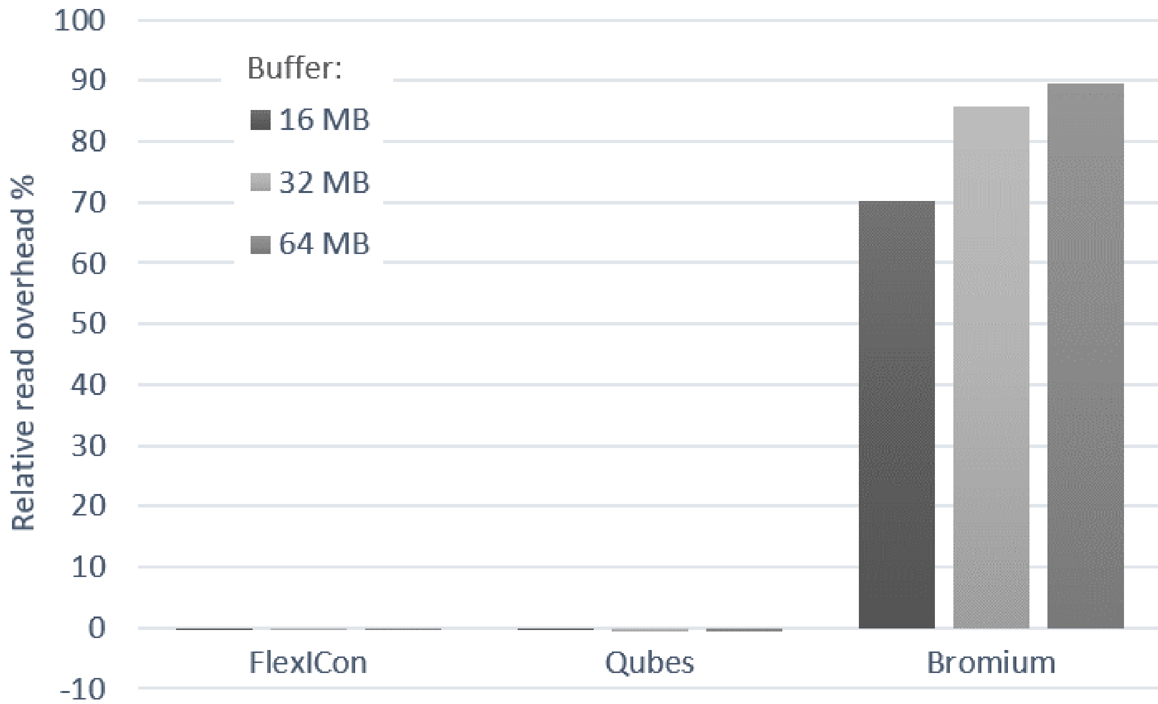


Figure 19. Disk read overhead

5.4.1 Disk Read Throughput

Disk read throughput was measured for 1 GB files. To avoid reading from the cache, 10 different files were read in succession. Since the total size (10 GB) is larger than the available RAM, the files will not fit in the cache. The following buffer sizes were tested: 16 MB, 32 MB and 64 MB. Each scenario was repeated 10 times. Figure 19 shows the relative overheads. FlexICon and Qubes have no overhead while Bromium has an 82% overhead. This large overhead is probably caused by a mechanism in Bromium that looks for suspicious activity in MicroVMs for reporting purposes.

Note that the baseline values for all platforms were very close, so it is meaningful to make a comparison based on the relative overheads shown in Figure 19. A box plot of the read throughput data is shown in Figure 20.

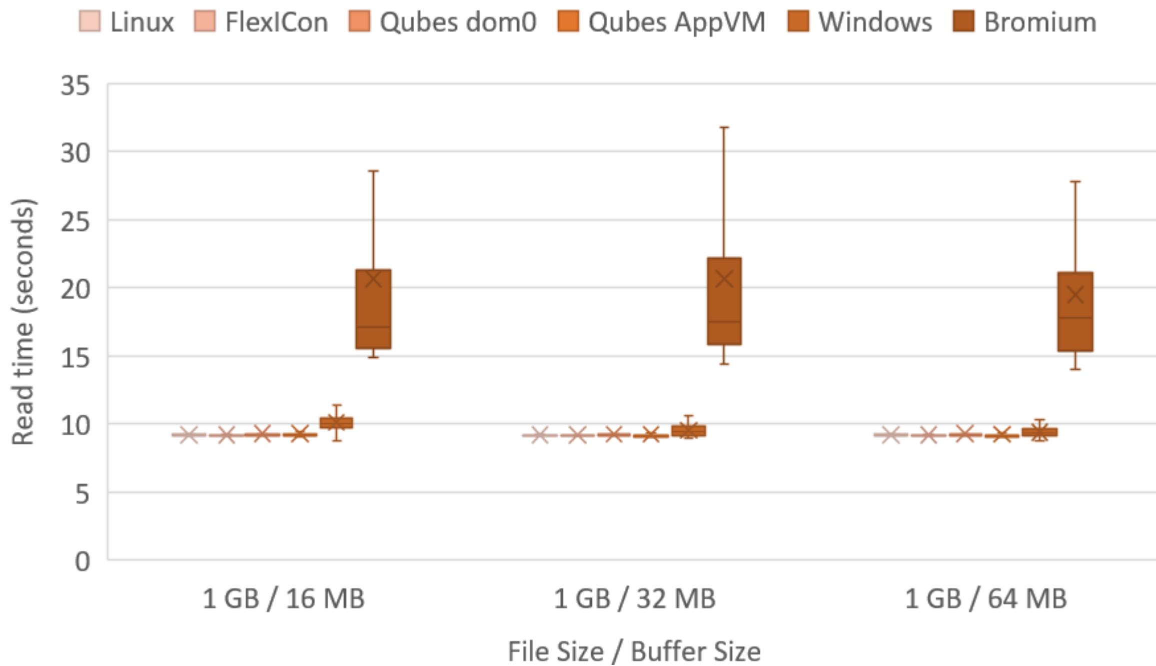


Figure 20. Box plot of disk read throughput data

5.4.2 Disk Write Speed

The time it takes to write files of various sizes was measured for all platforms. The following file sizes were tested: 1 KB, 16 KB, 128 KB, 1 MB, 16 MB, 128 MB and 1 GB. The entire contents of the file was generated in one buffer and written to the disk with one call to the `Write()` method. To ensure the file is fully written on the disk, a call to the `Sync()` method was made before closing the file. Figure 21 summarizes the calculated overheads for all file sizes. The results show no overhead for FlexICon as expected. Qubes shows higher overhead for smaller files, and Bromium shows higher overhead for larger files. Figures 22, 23, 24 and 25 show box plots of disk write speed data.

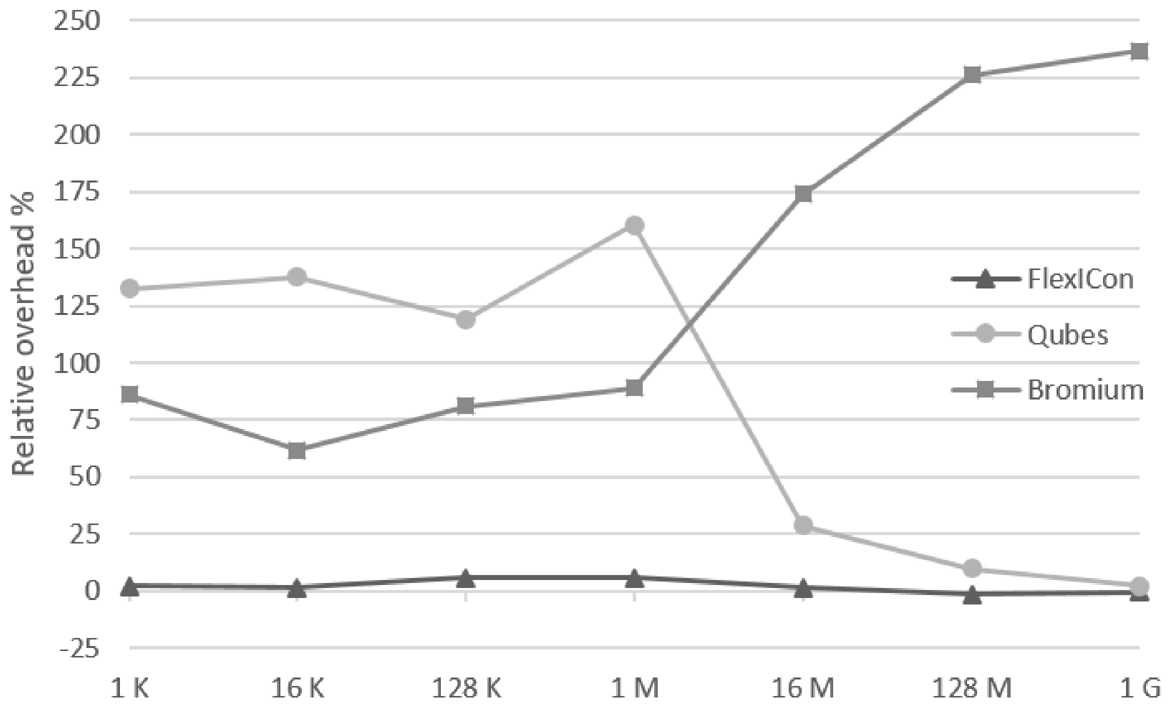


Figure 21. Disk write overhead

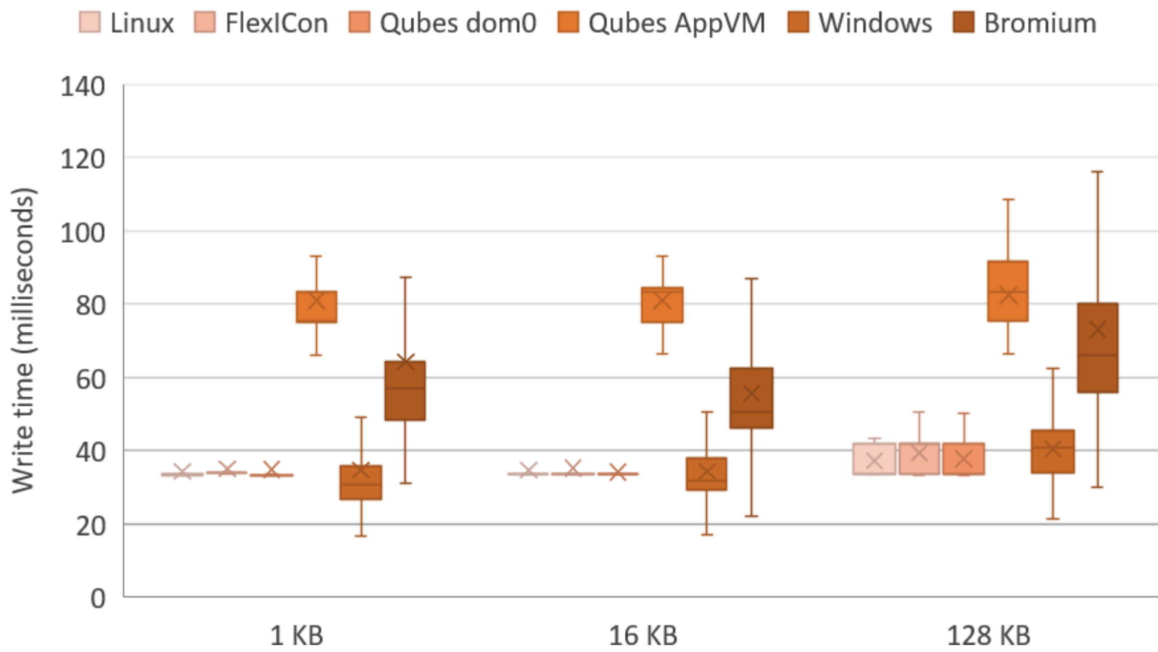


Figure 22. Box plot of disk write speed data for tiny files

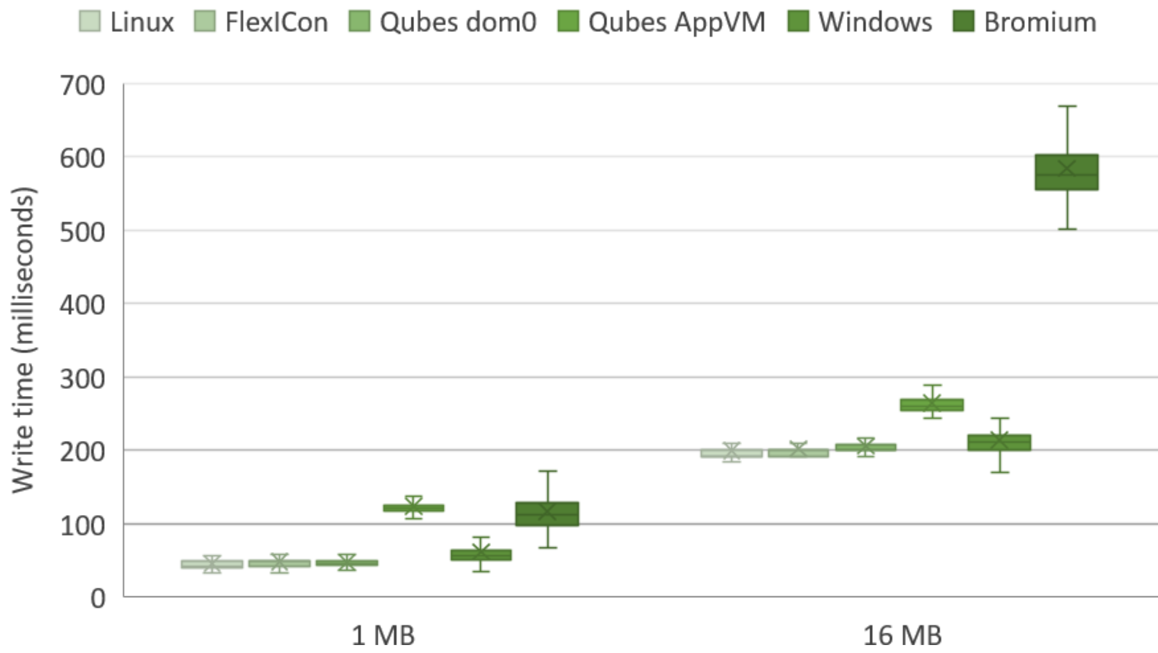


Figure 23. Box plot of disk write speed data for small files

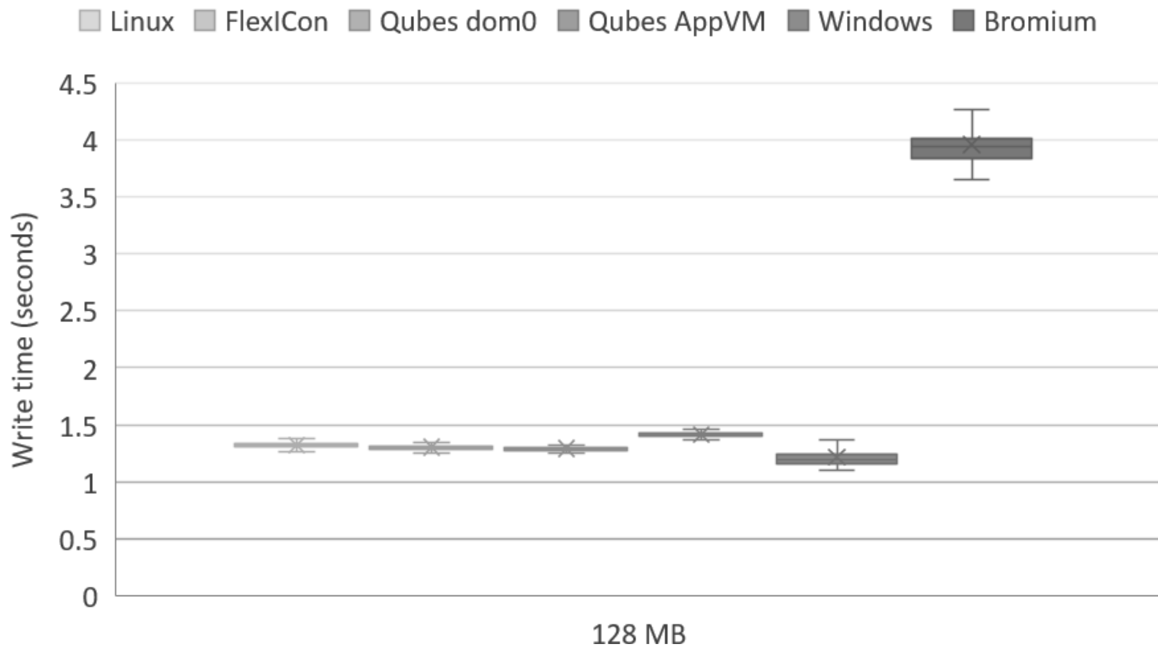


Figure 24. Box plot of disk write speed data for medium files

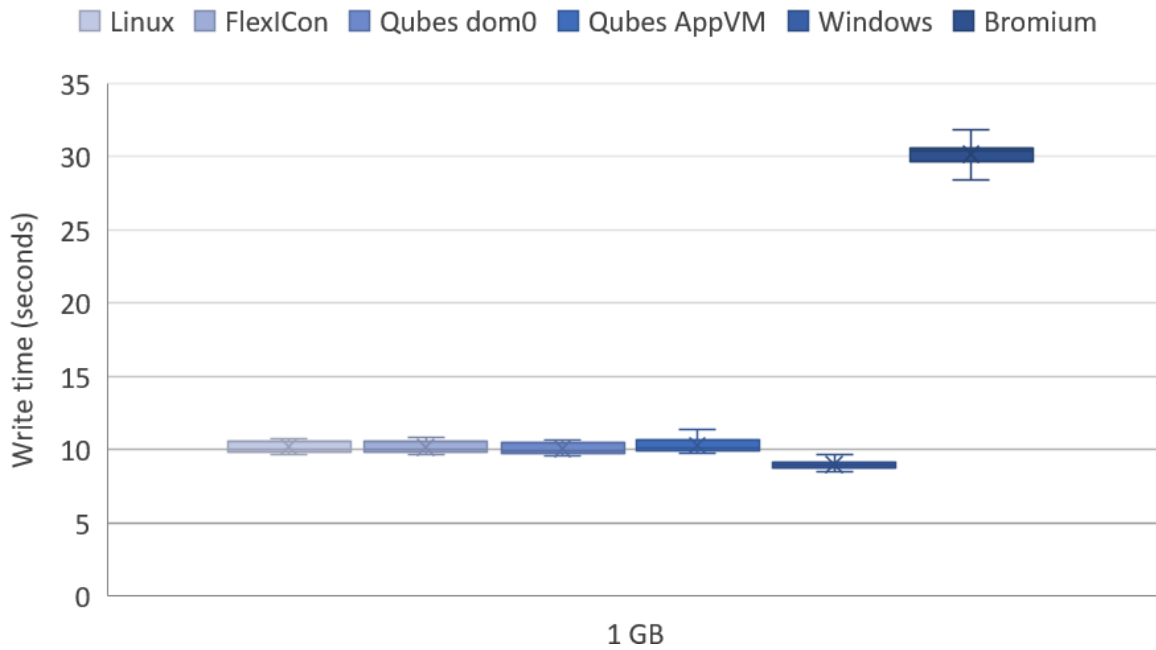


Figure 25. Box plot of disk write speed data for large files

5.4.3 Copying Files Between Containers

The time it takes to copy files of various sizes from one container to another was measured for both Qubes and FlexICon. Note that it is not possible to copy files between MicroVMs in Bromium. Qubes provides a command for copying files between containers (`qvm-copy-to-vm`). For FlexICon, `cp` followed by `sync` was used to copy files between containers. In both cases the measured time reflects the time it takes to fully commit the data to disk. Figure 26 shows the average time for copying files between containers. Qubes takes longer to copy files compared to FlexICon and the difference is more pronounced for smaller files.

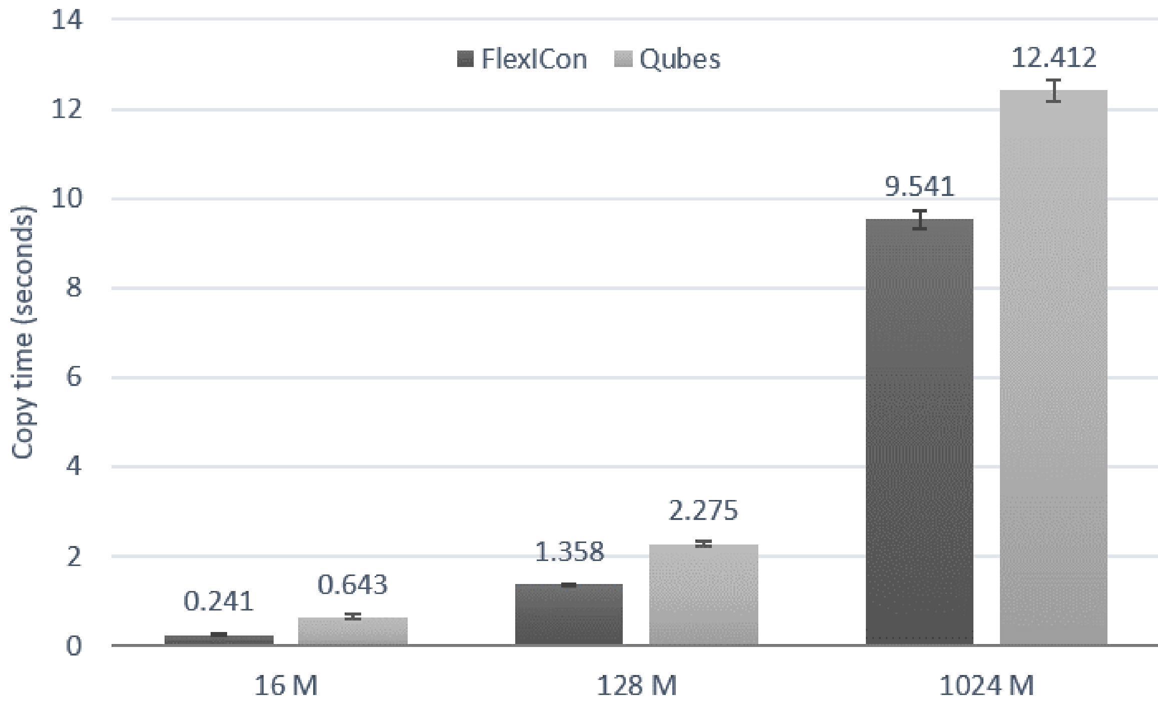


Figure 26. Copying files between containers

5.5 Network Access Overhead

To measure network throughput and latency, a 100 Mbps LAN with two computers was used to avoid network unpredictability problems. To calculate overhead, the following baselines were used:

- For FlexICon: native Linux was used as baseline
- For Qubes: sys-net VM was used as baseline since it has direct access to network hardware
- For Bromium: native Windows was used as baseline

5.5.1 Network Latency

To measure network latency, `lighttpd` was used to serve files of various sizes on the server and a program written in Go was used on the test machine to download the files. To account for the effect of concurrent network access, either 1, 2, 4 or 8 files were downloaded concurrently. The absolute overhead values are shown in Figures 27, 28, 29 and 30.

The results indicate that FlexICon has the lowest overhead among the three, while Bromium has the highest overhead which is probably affected by the monitoring system in Bromium that looks for suspicious activity in MicroVMs. It appears that the overhead does not scale proportionately with the file size, which means the relative overhead goes down with larger file sizes. For example, for FlexICon, the largest relative overhead is for 1 KB files downloaded serially (roughly 15%), and the smallest relative overhead is for 10 MB files ($\leq 0.01\%$).

5.5.2 Network Throughput

To measure network throughput, `iperf` version 3 was used. The goal was to find the maximum possible throughput for the test network setup on all platforms. iperf was set to execute for 5 minutes in each direction (upload and download) for all platforms. The results show that the throughput is not affected for any of the three systems. Table 4 summarizes the results.

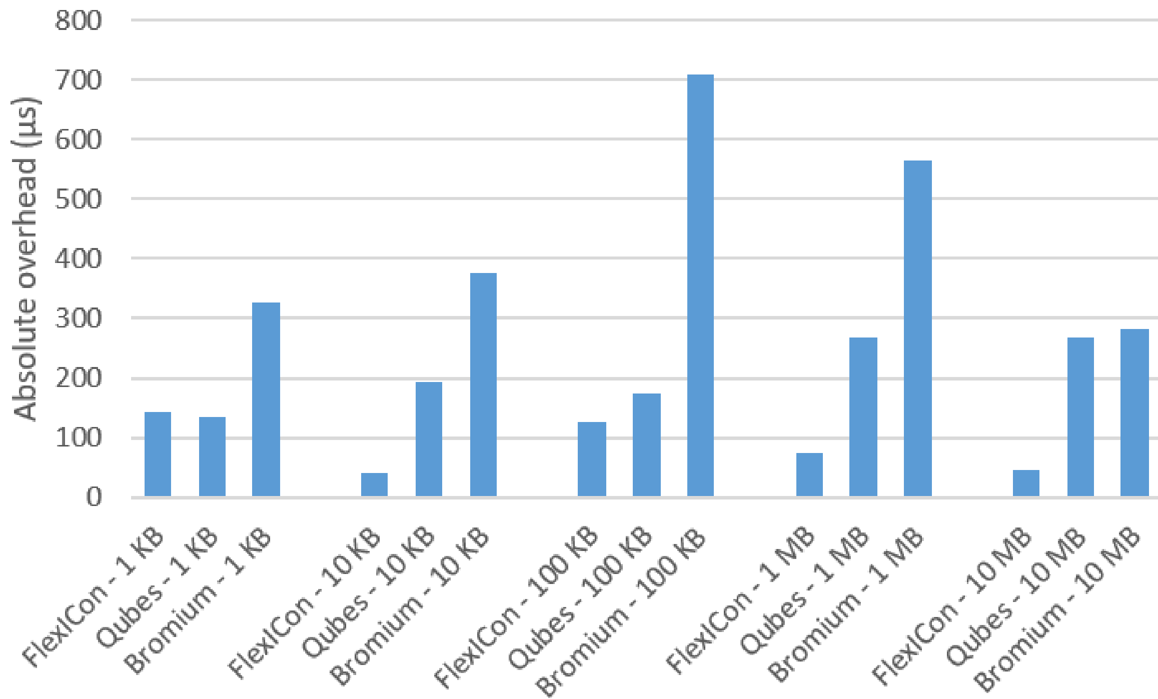


Figure 27. Latency overhead for serial downloads

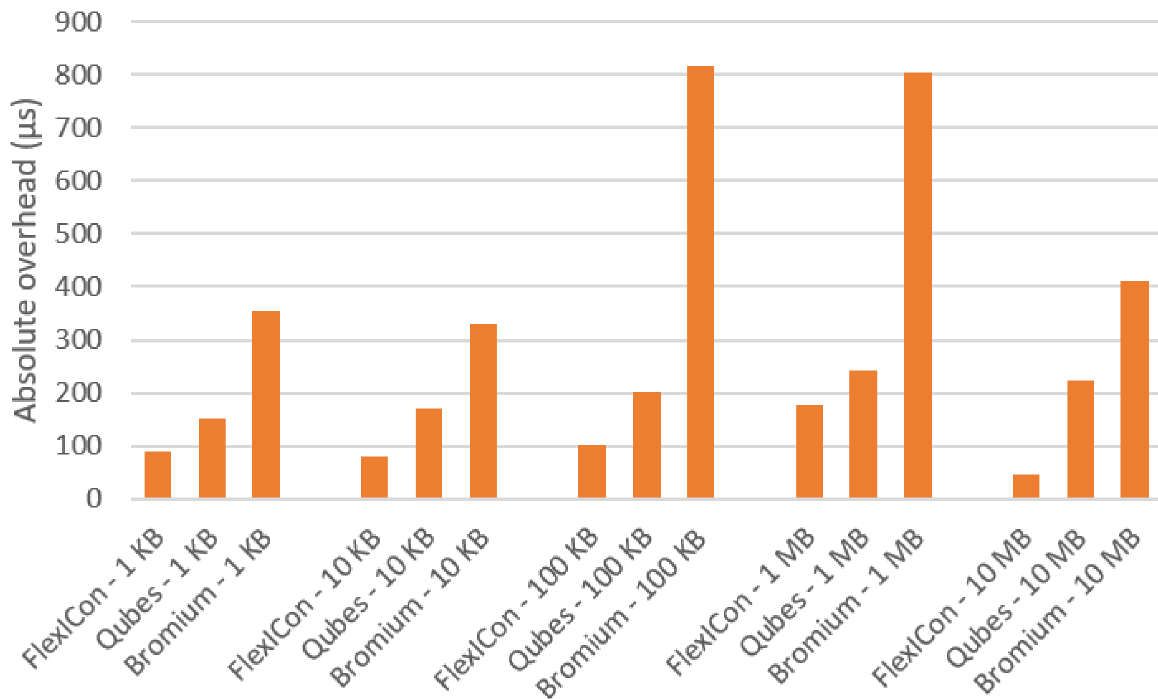


Figure 28. Latency overhead for 2 concurrent downloads

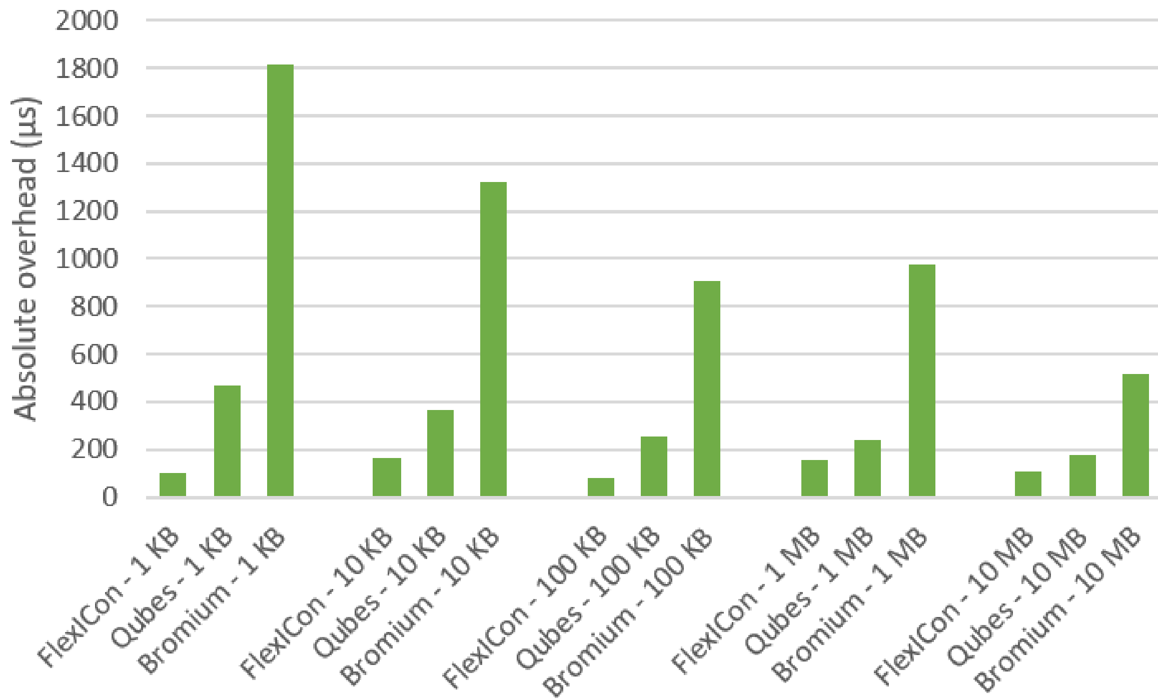


Figure 29. Latency overhead for 4 concurrent downloads

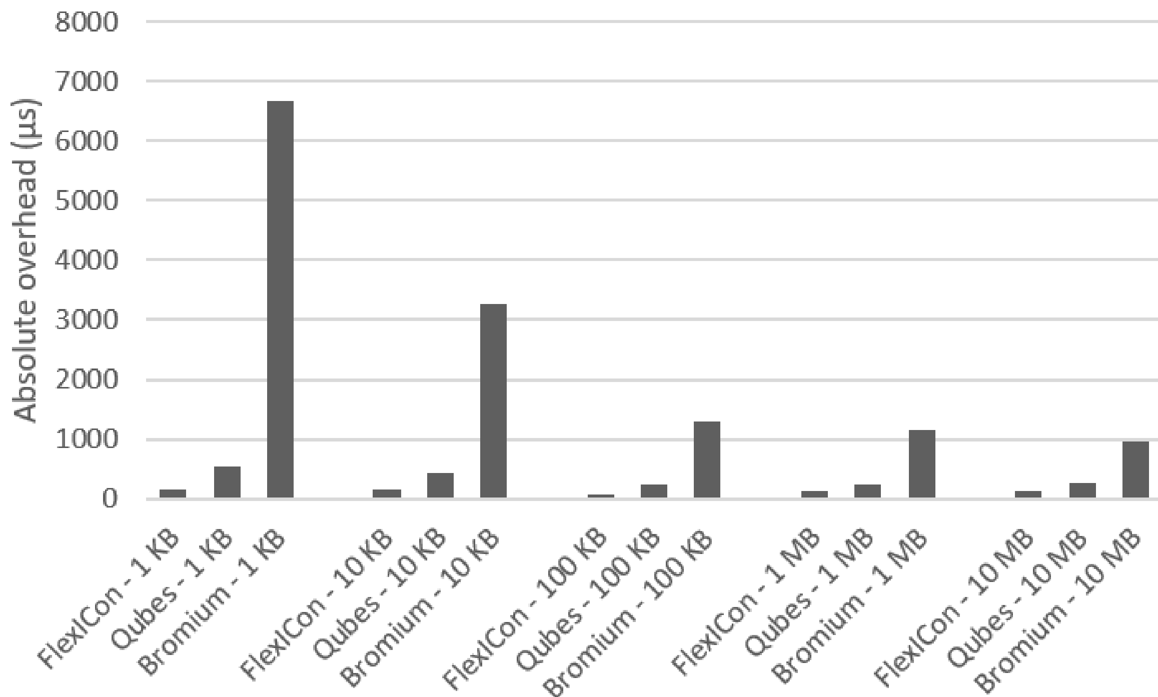


Figure 30. Latency overhead for 8 concurrent downloads

Table 4. Network throughput

Platform	Download	Upload
Native Linux	94.2 Mbps	94.0 Mbps
FlexICon	94.2 Mbps	94.0 Mbps
Native Windows	94.9 Mbps	94.7 Mbps
Bromium	94.9 Mbps	94.7 Mbps
Qubes sys-net	94.9 Mbps	94.7 Mbps
Qubes AppVM	94.9 Mbps	94.7 Mbps

5.6 GUI

Measuring the performance overhead of the GUI system is complicated due to the way the X server in the container and the X server running on the host interact with each other. There is not much feedback from the host to the container about what has been successfully drawn on the screen which means, for example, that frame rate statistics reported by a video player inside the container is not accurate. However, a qualitative analysis of the GUI performance and its limitations is possible. It should also be noted that the GUI performance of FlexICon would be similar to that of Qubes since the same model is used in both cases.

Qualitative analysis of the graphic user interface did not reveal any noticeable performance overhead when browsing the web, playing multiple videos or viewing documents. The reason is that the GUI module uses shared memory for rendering container windows, so rendering is at RAM speed and no copying is involved. On the other hand, 3D rendering in containers is performed by the CPU because the X server running in the container does not have access to the GPU. This means that 3D rendering in containers is not going to be fast.

Chapter 6

CONCLUSION & FUTURE WORK

This dissertation has shown that it is possible to automate the best practice of compartmentalizing access to web content in a mostly browser-independent manner without changing the browser codebase that can be ported to all major desktop operating systems. However, a limitation of this method is the imperfect isolation of browser instances which can be solved by moving to a more comprehensive compartmentalization solution.

This dissertation has shown that it is possible to have a compartmentalization solution that achieves high performance while providing significant improvement to the security of desktop systems by leveraging light-weight virtualization and providing cooperative features that reduce the number of security decisions made by the user at runtime. The work leaves open some important avenues for further improvements that are discussed in the following section.

6.1 Future Work

Possible avenues for future work on FlexICon include:

- Exploring ideas for creating the policy. For example, one could have a template policy that can be customized for every user through a tool that asks the user about his/her frequently visited websites in pre-defined categories. Alternatively, a learning method could be used to gather information about user's web activities to customize the template policy based on the gathered data.
- Devising an automated tool for identifying domain names that collectively form a

website. Such a tool can be used for defining the list of allowed domains for web-based containers.

- Further dividing each web-based container so that the browser runs in its own sub-container separate from other applications. This will help minimize the risk of malicious code interfering with browser operation including modification to browser network settings.
- Improving the implementation of FlexICon by:
  - Preventing association of domain names to private IP addresses which could be used in DNS rebinding attacks to gain access to unprotected local web consoles.
  - Implementing rate-limiting mechanism in the URL handling module to prevent compromised containers from opening too many URLs.
  - Implementing sound support in containers.

REFERENCES

[1] S. Abraham and I. Chengalur-Smith. "An overview of social engineering malware: Trends, tactics, and implications". In: *Technology in Society* 32.3 (2010), pp. 183–196.

[2] D. Akhawe and A. P. Felt. "Alice in Warningland: A Large-scale Field Study of Browser Security Warning Effectiveness". In: *Proceedings of the 22Nd USENIX Conference on Security*. SEC'13. Washington, D.C.: USENIX Association, 2013, pp. 257–272. URL: http://dl.acm.org/citation.cfm?id=2534766.2534789.

[3] J. Andrus et al. "Cells: A Virtual Mobile Smartphone Architecture". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 173–187. URL: http://doi.acm.org/10.1145/2043556.2043574.

[4] D. Atkins and R. Austein. *Threat Analysis of the Domain Name System (DNS)*. RFC 3833. Aug. 2004. URL: https://rfc-editor.org/rfc/rfc3833.txt.

[5] M. Balduzzi et al. "A Solution for the Automated Detection of Clickjacking Attacks". In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ASIACCS'10. Beijing, China, 2010, pp. 135–144.

[6] A. Barth. *The Web Origin Concept*. RFC 6454. RFC Editor, Dec. 2011. URL: https://tools.ietf.org/rfc/rfc6454.txt.

[7] A. Barth et al. *The Security Architecture of the Chromium Browser*. Tech. rep. 2008.

[8] A. Barth, C. Jackson, and J. C. Mitchell. "Robust Defenses for Cross-site Request Forgery". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS'08. Alexandria, Virginia, USA, 2008, pp. 75–88.

[9] T. Berners-Lee, R. T. Fielding, and L. M. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Jan. 2005. URL: https://tools.ietf.org/rfc/rfc3986.txt.

[10] E. W. Biederman and L. Networx. "Multiple Instances of the Global Linux Namespaces". In: *Proceedings of the Linux Symposium*. Vol. 1. 2006, pp. 101–112.

[11] D. F. C. Brewer and M. J. Nash. "The Chinese Wall Security Policy". In: *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*. IEEE. 1989, pp. 206–214.

[12] *Bromium vSentry*. DS.VSENTRY.US-EN.1512. Datasheet DS.VSENTRY.US-EN.1512. Bromium Inc. 2015.

[13] R. Capizzi et al. “Preventing Information Leaks Through Shadow Executions”. In: *Proceedings of the 2008 Annual Computer Security Applications Conference*. ACSAC ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 322–331. URL: http://dx.doi.org/10.1109/ACSAC.2008.50.

[14] US-CERT. *Securing Your Web Browser*. Retrieved February 2017 from https://www.us-cert.gov/publications/securing-your-web-browser.

[15] K. Chandrasekar et al. *Internet Security Threat Report*. Tech. rep. Symantec, 2017. URL: https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf.

[16] C. Chaubal. “The Architecture of VMware ESXi”. In: *VMware White Paper* 1.7 (2008).

[17] J. Che et al. “A Synthetical Performance Evaluation of OpenVZ, Xen and KVM”. In: *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*. 2010, pp. 587–594.

[18] S. Checkoway et al. “Return-oriented Programming Without Returns”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS ’10. Chicago, Illinois, USA: ACM, 2010, pp. 559–572. URL: http://doi.acm.org/10.1145/1866307.1866370.

[19] E. Y. Chen et al. “App Isolation: Get the Security of Multiple Browsers with Just One”. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS’11. Chicago, Illinois, USA, 2011, pp. 227–238.

[20] Chromium Project. *Benchmarking Extension*. Retrieved March 2018 from https://www.chromium.org/developers/design-documents/extensions/how-the-extension-system-works/chrome-benchmarking-extension.

[21] R. S. Cox et al. “A Safety-oriented Platform for Web Applications”. In: *Security and Privacy, 2006 IEEE Symposium on*. IEEE. May 2006.

[22] L. F. Cranor. “A Framework for Reasoning About the Human in the Loop”. In: *Proceedings of the 1st Conference on Usability, Psychology, and Security*. UPSEC’08. San Francisco, California: USENIX Association, 2008, 1:1–1:15. URL: http://dl.acm.org/citation.cfm?id=1387649.1387650.

[23] A. Czeskis et al. “Lightweight Server Support for Browser-based CSRF Protection”. In: *Proceedings of the 22Nd International Conference on World Wide Web*. WWW ’13. Rio de Janeiro, Brazil: ACM, 2013, pp. 273–284. URL: http://doi.acm.org/10.1145/2488388.2488413.

[24] D. Dai Zovi. “Practical return-oriented programming”. In: *SOURCE Boston* (2010).

[25] W. De Groef et al. “FlowFox: a web browser with flexible and precise information flow control”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 748–759.

[26] J. Epstein. “Fifteen Years after TX: A Look Back at High Assurance Multi-Level Secure Windowing”. In: *Computer Security Applications Conference, 2006. ACSAC ’06. 22nd Annual*. 2006, pp. 301–320.

[27] J. Epstein and M. Shugerman. “A Trusted X Window System Server for Trusted Mach”. In: *USENIX MACH Symposium*. 1990, pp. 141–156.

[28] J. Epstein et al. “A High Assurance Window System Prototype”. In: *Journal of Computer Security* 2.2-3 (1993), pp. 159–190.

[29] N. Feske and C. Helmuth. “A Nitpicker’s Guide to a Minimal-complexity Secure GUI”. In: *Proceedings of the 21st Annual Computer Security Applications Conference*. ACSAC’05. 2005, pp. 85–94.

[30] B. Ford and R. Cox. “Vx32: Lightweight User-level Sandboxing on the x86”. In: *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. ATC’08. Boston, Massachusetts: USENIX Association, 2008, pp. 293–306. URL: http://dl.acm.org/citation.cfm?id=1404014.1404039.

[31] E. Gandotra, D. Bansal, and S. Sofat. “Zero-day malware detection”. In: *Embedded Computing and System Design (ISED), 2016 Sixth International Symposium on*. IEEE. 2016, pp. 171–175.

[32] T. Garfinkel. “Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools”. In: *Proceedings of the 2003 Network and Distributed Systems Security Symposium*. Vol. 3. NDSS’03. 2003, pp. 163–176.

[33] T. Garfinkel, B. Pfaff, and M. Rosenblum. “Ostia: A Delegating Architecture for Secure System Call Interposition”. In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*. NDSS. The Internet Society, 2004.

[34] A. Gazet. “Comparative analysis of various ransomware virii”. In: *Journal in Computer Virology* 6.1 (2010), pp. 77–90. URL: http://dx.doi.org/10.1007/s11416-008-0092-2.

[35] C. Greamo and A. Ghosh. “Sandboxing and Virtualization: Modern Tools for Combating Malware”. In: *IEEE Security and Privacy* 9.2 (Mar. 2011), pp. 79–82. URL: http://dx.doi.org/10.1109/MSP.2011.36.

[36] E. Hammer-Lahav and B. Cook. *Web Host Metadata*. Oct. 2011. URL: https://tools.ietf.org/rfc/rfc6415.txt.

[37] C. Heffner. “Remote Attacks Against SOHO Routers”. In: *Blackhat USA* (2010).

[38] A. E. Howe et al. “The Psychology of Security for the Home Computer User”. In: *2012 IEEE Symposium on Security and Privacy*. May 2012, pp. 209–223.

[39] L.-S. Huang et al. “Clickjacking: Attacks and Defenses”. In: *USENIX Security Symposium*. 2012, pp. 413–428.

[40] G. P. Im and R. L. Baskerville. “A Longitudinal Study of Information System Threat Categories: The Enduring Problem of Human Error”. In: *SIGMIS Database* 36.4 (Oct. 2005), pp. 68–79. URL: http://doi.acm.org/10.1145/1104004.1104010.

[41] I. Ion, R. Reeder, and S. Consolvo. ““... No one Can Hack My Mind”: Comparing Expert and Non-Expert Security Practices”. In: *2015 Symposium on Usable Privacy and Security*. SOUPS’15. 2015, pp. 327–346.

[42] C. Jackson et al. “Protecting Browsers from DNS Rebinding Attacks”. In: *ACM Transactions on the Web (TWEB)* 3.1 (2009), p. 2.

[43] M. Johns. “On JavaScript Malware and related threats”. In: *Journal in Computer Virology* 4.3 (2008), pp. 161–178.

[44] M. Johns, B. Engelmann, and J. Posegga. “XSSDS: Server-side Detection of Cross-site Scripting Attacks”. In: *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. IEEE. 2008, pp. 335–344.

[45] T. Kim and N. Zeldovich. “Practical and Effective Sandboxing for Non-root Users”. In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC’13. San Jose, CA: USENIX Association, 2013, pp. 139–144. URL: http://dl.acm.org/citation.cfm?id=2535461.2535478.

[46] M. Krasnyansky and M. Yevmenkin. *Universal TUN/TAP device driver*. Retrieved June 2016 from https://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[47] H. M. Levy. *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984.

[48] Linux Containers. *LXC: Linux Containers*. Retrieved June 2016 from https://linuxcontainers.org/.

[49] Linux Foundation. *Xen Project*. Retrieved January 2018 from https://www.xenproject.org/.

[50] Linux man-pages. *ptrace(2) Linux Programmer's Manual*. Retrieved March 2018 from http://man7.org/linux/man-pages/man2/ptrace.2.html.

[51] Linux man-pages. *syscalls(2) Linux Programmer's Manual*. Retrieved March 2018 from http://man7.org/linux/man-pages/man2/syscalls.2.html.

[52] P. K. Manadhata and J. M. Wing. "An Attack Surface Metric". In: *IEEE Transactions on Software Engineering* 37.3 (May 2011), pp. 371–386.

[53] M. S. Miller, K.-P. Yee, J. Shapiro, et al. *Capability Myths Demolished*. Tech. rep. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www.erights.org/elib/capability/duals, 2003.

[54] A. Moshchuk, H. J. Wang, and Y. Liu. "Content-based Isolation: Rethinking Isolation Policy Design on Client Systems". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS'13. Berlin, Germany, 2013, pp. 1167–1180. URL: http://doi.acm.org/10.1145/2508859.2516722.

[55] R. O'Leary. *Application Security Statistics Report*. Tech. rep. WhiteHat Security, 2017.

[56] OpenVZ Community. *OpenVZ*. Retrieved March 2018 from http://openvz.org/.

[57] OWASP. *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet*. Retrieved February 2018 from https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

[58] N. Provos et al. "The Ghost in the Browser Analysis of Web-based Malware". In: *Proceedings of the First Workshop on Hot Topics in Understanding Botnets*. HotBots'07. Cambridge, MA, 2007, pp. 4–4.

[59] F. Roesner et al. "User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems". In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP'12. 2012, pp. 224–238.

[60] J. Rutkowska and R. Wojtczuk. “Qubes OS Architecture”. In: *Invisible Things Lab Tech Rep* (2010).

[61] R. W. Scheifler and J. Gettys. “The X Window System”. In: *ACM Trans. Graph.* 5.2 (Apr. 1986), pp. 79–109.

[62] M. Shahzad, M. Z. Shafiq, and A. X. Liu. “A Large Scale Exploratory Analysis of Software Vulnerability Life Cycles”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE’12. Zurich, Switzerland: IEEE Press, 2012, pp. 771–781. URL: http://dl.acm.org/citation.cfm?id=2337223.2337314.

[63] J. S. Shapiro, J. M. Smith, and D. J. Farber. “EROS: A Fast Capability System”. In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP’99. Charleston, South Carolina, USA, 1999, pp. 170–185.

[64] J. S. Shapiro et al. “Design of the EROS Trusted Window System”. In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM’04. San Diego, CA, 2004, pp. 12–12.

[65] A. Soltani et al. “Flash Cookies and Privacy.” In: *AAAI Spring Symposium: Intelligent Information Privacy Management*. Vol. 2010. 2010, pp. 158–163.

[66] M. Souppaya and K. Scarfone. “Guide to Malware Incident Prevention and Handling for Desktops and Laptops”. In: *NIST special publication* 800–83 Rev. 1 (2013). URL: http://doi.org/10.6028/NIST.SP.800-83r1.

[67] M. Stiegler et al. “Polaris: Virus-safe Computing for Windows XP”. In: *Commun. ACM* 49.9 (Sept. 2006), pp. 83–88.

[68] W. K. Sze and R. Sekar. “Provenance-based Integrity Protection for Windows”. In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACSAC’15. Los Angeles, CA, USA, 2015, pp. 211–220.

[69] W. Vogels. “Beyond Server Consolidation”. In: *Queue* 6.1 (Jan. 2008), pp. 20–26.

[70] D. A. Wagner. “Janus: an Approach for Confinement of Untrusted Applications”. CSD-99-1056. MA thesis. Berkeley, CA, USA: UC Berkeley, 1999.

[71] R. N. M. Watson et al. “Capsicum: Practical Capabilities for UNIX”. In: *Proceedings of the 19th USENIX Conference on Security*. USENIX Security’10. Washington, DC, 2010, pp. 3–3. URL: http://dl.acm.org/citation.cfm?id=1929820.1929824.

[72] *Content Security Policy Level 3*. W3C Working Draft. W3C, Sept. 2016. URL: https://www.w3.org/TR/2016/WD-CSP3-20160913/.

[73] M. West and M. Goodwin. *Same-Site Cookies*. Internet-Draft draft-ietf-httpbis-cookie-same-site-00. IETF Secretariat, June 2016. URL: https://tools.ietf.org/id/draft-ietf-httpbis-cookie-same-site-00.txt.

[74] B. Yee et al. “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”. In: *Commun. ACM* 53.1 (Jan. 2010), pp. 91–99. URL: http://doi.acm.org/10.1145/1629175.1629203.

[75] K.-P. Yee. “Aligning Security and Usability”. In: *IEEE Security and Privacy* 2.5 (Sept. 2004), pp. 48–55.

[76] K.-P. Yee. “Secure Interaction Design and the Principle of Least Authority”. In: *Proceedings of the CHI Workshop on Human-Computer Interaction and Security Systems*. 2003.

[77] M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

APPENDIX A

SAMPLE FLEXICON POLICY

The following is a sample FlexICon configuration file that defines the following containers:

- Banking: dedicated to online banking
- Shopping: dedicated to online shopping
- Internet: for general Internet browsing
- User: for running certain applications when launched directly
- FileBrowser: for running the file browser application
- Junk: for running untrusted applications
- Email: for running the email client
- EmailDisposable: for opening email attachments

Note that the provided configuration is meant as an illustrative example, but it can be extended for real use by including more applications, domain names and URLs.

```
# Define application paths
define FIREFOX "/usr/bin/firefox"
define KATE "/usr/bin/kate"
define DOLPHIN "/usr/bin/dolphin"
define KMAIL "/usr/bin/kmail"

define BROWSERS { + FIREFOX, - "*" }

# Domains needed for the Banking container
define BANKING_DOMAINS [
  "[*.]usbank.com",
  "[*.]bankofamerica.com",
  "[*.]bac-assets.com"
]

# Banking websites
define BANKING_URLS [
  "^://[^.]usbank.com[/*]",
  "^://[^.]bankofamerica.com[/*]"
]

# Domains needed for the Shopping container
define SHOPPING_DOMAINS [
  "[*.]amazon.com",
  "[*.]awsstatic.com",
  "[*.]images-amazon.com",
  "[*-]images-amazon.com",
  "[*.]target.com",
  "[*.]targetimg1.com",
  "[*.]targetimg2.com",
  "[*.]targetimg3.com",
  "[*.]bestbuy.com",
  "[*.]bbystatic.com",
  "[*.]bby.com"
]
```

```
# Shopping websites
define SHOPPING_URLS [
  "^://[^.]amazon.com[/*]",
  "^://[^.]target.com[/*]",
  "^://[^.]bestbuy.com[/*]"
]

# Certificate authorities and other common domains
define COMMON_DOMAINS [
  "[*.]verisign.com",
  "[*.]comodo.com",
  "[*.]digicert.com",
  "[*.]entrust.net",
  "[*.]globalsign.com",
  "[*.]sb-ssl.google.com",
  "[*.]akamaiedge.net",
  "[*.]edgekey.net",
  "[*.]amazonaws.com",
  "[*.]demdex.net",
  "[*.]cloudfront.net",
  "[*.]akamai.net",
  "[*.]symcd.com"
]

define ANY { + "*" }
define ALL { + "*" }
define NONE { - "*" }

# Container definitions

container "Banking" {
    color_index: 4
    gui: true
    system_files: normal
    home_extras: []
    save_as_other_containers: []
    read_from_other_containers: []
```

```
    network_access: true
    domains: {
        + COMMON_DOMAINS,
        + BANKING_DOMAINS,
        - "*"
    }
    usr_bin: ALL
}

container "Shopping" {
    color_index: 3
    gui: true
    system_files: normal
    home_extras: []
    save_as_other_containers: []
    read_from_other_containers: []
    network_access: true
    domains: {
        + COMMON_DOMAINS,
        + SHOPPING_DOMAINS,
        - "*"
    }
    usr_bin: ALL
}

container "Internet" {
    color_index: 1
    gui: true
    system_files: normal
    home_extras: []
    save_as_other_containers: []
    read_from_other_containers: []
    network_access: true
    domains: {
        - BANKING_DOMAINS,
        - SHOPPING_DOMAINS,
        + "*"
```

```
    }
    usr_bin: ALL
}

container ”User” {
    color_index: 8
    gui: true
    system_files: normal
    home_extras: []
    save_as_other_containers: [
        ”Internet”,
        ”Banking”,
        ”Shopping”
    ]
    read_from_other_containers: []
    network_access: false
    domains: NONE
    usr_bin: ALL
}

container ”FileBrowser” {
    color_index: 6
    gui: true
    system_files: normal
    home_extras: [
        ”/Banking”,
        ”/Shopping”,
        ”/Internet”,
        ”/User”,
        ”/Junk”
    ]
    save_as_other_containers: []
    read_from_other_containers: []
    network_access: false
    domains: NONE
    usr_bin: {
        # Prevent these apps from executing in FileBrowser,
```

```
        # so that when the user clicks a text or PDF file
        # in dolphin, it is opened in the proper container.
        - ”kate”,
        - ”okular”,
        + ”*”
    }
}

# Junk container is used to execute untrusted apps
container ”Junk” {
    color_index: 1
    gui: true
    system_files: normal
    home_extras: []
    save_as_other_containers: []
    read_from_other_containers: []
    network_access: false
    domains: NONE
    usr_bin: ALL
}

container ”Email” {
    color_index: 5
    gui: true
    system_files: normal
    home_extras: []
    save_as_other_containers: []
    read_from_other_containers: [
        ”Banking”,
        ”Shopping”,
        ”Internet”,
        ”User”
    ]
    network_access: true
    domains: ALL
    usr_bin: {
        - ”kate”,
```

```
        - "okular",
        - "firefox",
        + "*"
    }
}

# EmailDisposable is used to open email attachment files
disposable "EmailDisposable" {
    color_index: 1
    gui: true
    system_files: normal
    home_extras: []
    save_as_other_containers: []
    read_from_other_containers: []
    network_access: false
    domains: NONE
    usr_bin: ALL
}

# Rules

# Run the Dolphin file browser in FileBrowser
if {
    application {
        + DOLPHIN,
        - "*"
    }
} then "FileBrowser"

# Run KMail in Email
if {
    application {
        + KMAIL,
        - "*"
    }
} then "Email"
```

```
# Open Banking websites in Banking
if {
    application BROWSERS
    url {
        + BANKING_URLS,
        - "*"
    }
} then "Banking"

# Open Banking files in Banking
if {
    application ANY
    file {
        + "/Banking/*",
        - "*"
    }
} then "Banking"

# Open Shopping websites in Shopping
if {
    application BROWSERS
    url {
        + SHOPPING_URLS,
        - "*"
    }
} then "Shopping"

# Open Shopping files in Shopping
if {
    application ANY
    file {
        + "/Shopping/*",
        - "*"
    }
} then "Shopping"

# Open other websites in Internet
```

```
if {
    application BROWSERS
    url {
        - BANKING_URLS,
        - SHOPPING_URLS,
        + "*"
    }
} then "Internet"

# Open Internet files in Internet
if {
    application ANY
    file {
        + "/Internet/*",
        - "*"
    }
} then "Internet"

# Open User files in User
if {
    application ANY
    file {
        + "/User/*",
        - "*"
    }
} then "User"

# Open Junk files in Junk
if {
    application ANY
    file {
        + "/Junk/*",
        - "*"
    }
} then "Junk"

# Open Email files in EmailDisposable
```

```
if {
    application ANY
    file {
        + "/Email/*",
        - "*"
    }
} then "EmailDisposable"

# Browsers launched directly should be opened in Internet
if { application BROWSERS } then "Internet"

# Certain apps launched directly should be opened in User
if {
  application {
    + KATE,
    - "*"
  }
} then "User"

# Any other applications should be opened in Junk
if { application ANY } then "Junk"
```