Title
Data Tethers: Preventing Information Leakage by Enforcing Environmental Data Access Policies

Author
Fleming, Charles

Publication Date
2013

Peer reviewed|Thesis/dissertation

University of California

Los Angeles

Data Tethers: Preventing Information Leakage by Enforcing Environmental Data Access Policies

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Charles Fleming

2013

Abstract of the Dissertation

Data Tethers: Preventing Information Leakage by Enforcing Environmental Data Access Policies

by

Charles Fleming

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2013

Professor Peter Reiher, Co-chair

Professor Todd Millstein, Co-chair

Protecting data from accidental loss or theft is crucial in today's world of mobile computing. Data Tethers provides environmental policy control of information at the data level, rather than the file level. Data Tethers uses dynamic recompilation to add label tracking instructions to existing applications in the OpenSolaris operating system, allowing fine-grain data flow tracking in legacy applications without the need to recompile from source. We demonstrate the system's feasibility with microbenchmarks that show individual component performance and benchmarks of real user applications like word processors and spreadsheets.

The dissertation of Charles Fleming is approved.

Jens Palsberg

William Kaiser

Mario Gerla

Todd Millstein, Committee Co-chair

Peter Reiher, Committee Co-chair

University of California, Los Angeles

2013

Table of Contents

List of Figures

List of Tables

Vita

1997 B.S. Mathematics, University of Southern Mississippi

CHAPTER 1

Introduction

Data loss is a serious concern for individuals, companies, institutions, and government agencies. While some data loss is caused by hacker intrusions, many large-scale data losses are caused by the physical loss of a portable computer or data storage device, or by accidental leakage by mis-configured software. Hundreds of thousands of sensitive records can be lost instantly when a laptop disappears from a coffee shop or a flash drive falls out of a bag, or when a sensitive directory is shared via peer to peer software.

Many organizations respond to this problem by mandating full-disk encryption for portable devices or data leakage prevention services, which typically scan outgoing data for sensitive information. These technologies help, but are not complete solutions. Full disk encryption does not help when a running laptop is stolen, or when the password that unlocks the encryption is guessable. Data leakage prevention cannot work when the data is already encrypted or in some forms of compression. Carelessness and shallow understanding of how computers really work make it hard to ensure that data is not lost.

The Data Tethers approach to this problem is based on the key observation that what is really desired by these organization is that *data is simply not accessible outside a safe environment.* The Data Tethers system allows organizations

to specify clearly and precisely what conditions they consider to be safe, and attach this policy to the data. It then provides a practical mechanism to enforce this policy, and to make sure that all future copies of the data, including both copies on disk and copies sent over the network, are also restricted by this policy. The net result is that data that is stolen or lost is protected because it cannot be accessed outside of the safe environment, which the would-be thief does not have access to. Because we also protect copies, data that leaves the user's machine, for example via portable media or sent via e-mail, is also secure.

1.1 The Mobile Data Loss Problem

The Ponemon Institute estimated in 2010 that one out of ten corporate laptops will be lost or stolen in their lifetime, with an estimated cost of $49,000 per lost device[41]. Despite the severity of the problem, there are no signs that it is going away. In fact, loss of health care data in the US increased 525% since 2009, mainly due to lost devices and portable media[14]. Secure data has also been mistakenly posted to websites[55], accidentally shared via peer-to-peer software[12], and lost on disposed backup tapes[14]. In addition to the cost of the lost data, these incidents are both highly embarrassing for the entities involved, and frequently result in lawsuits.

Mobile computing has exploded in recent years. First laptops, then netbooks, and now tablets and smart phones are replacing traditional desktops for many tasks. By the end of 2012, it's predicted that there will be more mobile devices in use than people on the planet, and by 2016 there will be 1.4 devices in use per person. The amount of data used by these devices increased by a factor of

2.3, year over year, which marks a fourth straight year of more than 2x increase of data usage[18]. Clearly both the number and amount of use these devices are seeing is growing rapidly.

In addition to raw numbers, the power of mobile devices is increasing as well. In table 1.1 we can see the first popular modern smart phone, the original iPhone, compared with the a modern phone, set to be released mid-2012 [68][69]. As you can see, the hardware for these devices is growing quickly in both power and usability. The main drive for this growth is the desire to more frequently do more complex tasks on the go.

Table 1.1: Comparison of original iPhone with newer phones

	iPhone 1	Xiaomi 2
Display	480 x 320	1280 x 720
CPU	412 Mhz single core	1.5Ghz quad core
RAM	128 Mb	2Gb
Storage	4Gb	32 Gb

With this new found mobility and more work being done on mobile devices, mobile data security looms as a major issue that needs to be solved. While data security has been around almost since the beginning of computing, mobile computing faces one particular issue that desktop or server environments do not: the devices themselves can be lost or stolen, often while still running. Desktops or servers may be stolen, but this is much rarer, and most security schemes for these types of machines assume that the attacker does not have physical access to the machine. This is exactly the opposite case for mobile devices, where the attacker almost always has access to the device. Given the scope of the current

problems with mobile data loss, and the explosion of mobile devices, this is a problem that must be solved.

1.2 The Data Tethers Approach

The Data Tethers approach is to attach *environmental policies* to sensitive data. These policies specify under what conditions the data may be accessed, and the data is kept in encrypted when those conditions are not met. Keys are stored remotely and are only available on the user's system when the environment is secure, and destroyed when it is not secure.

One problem with labeling data is that policies may be lost when applications create new files. Taint tracking, a traditional solution for this problem, attaches a label to data and tracks all applications that touch the data, using the labels to enforce a policy. Unfortunately, unless specially written programs are used, the taint must be applied at the process level to all files written by that program, whether they contain labeled data or not. This generally results in large numbers of unrelated files being tainted, making the system unusable. The Data Tethers system uses dynamic code rewriting techniques to allow label tracking in applications so that files are only tainted when they have labeled data written to them.

The Data Tethers system is composed of three major components which can be seen in Figure 1.1: the *policy server*, the *policy monitor*, the *dynamic recompiler*, plus modifications to the operating system. The policy server is an application that runs in a secure location and stores policies and keys for the data. The policy monitor runs on the local machine, monitoring environmental

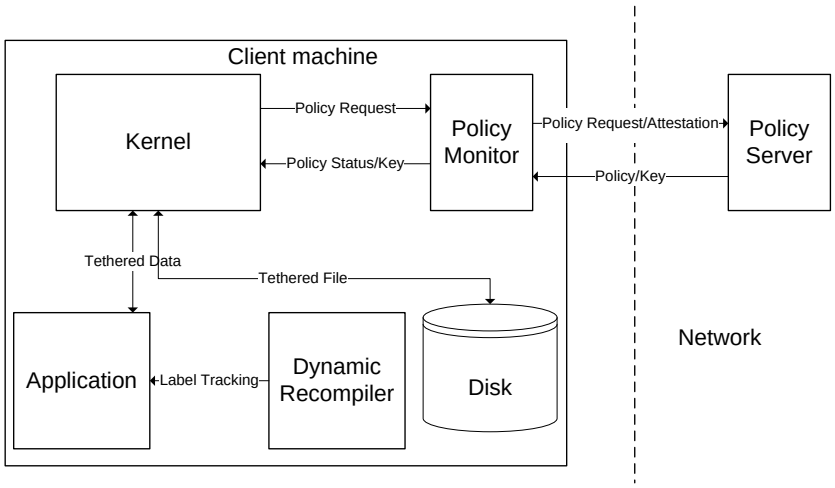

Figure 1.1: Data Tethers high-level architecture.

conditions, and handling communication with the policy server. The dynamic recompiler also runs locally and instruments applications that use tethered data. OS modifications support efficient tracking of data and handle policy violations.

Sensitive data is labeled with tags at the word level. When a user application reads this data from the disk, network, etc., the operating system detects that tethered data is being accessed. If the relevant policy is not currently being monitored, the policy monitor retrieves the policy from the policy server and verifies that the policy conditions are met. If so, the policy monitor retrieves the encryption key for the policy from the remote policy server, passes this key to the operating system, and adds the policy to the list of currently monitored policies. The operating system starts the dynamic recompiler for the application accessing the data. The dynamic recompiler adds code to the application to track the data labels. Labeled data written to a file by the application is encrypted with the proper encryption key and tagged with the policy ID. The policy monitor periodi-

cally reevaluates the policy at policy-specified intervals. When a violation occurs, it notifies the operating system, which suspends all processes using affected data, encrypts that data, and destroys the local encryption keys.

Data Tethers is not intended to be a replacement for the standard OpenSolaris access control mechanisms, and access to files is always contingent on the appropriate user permissions on the local machine.

An OpenSolaris based implementation of the system is presented with a description of modifications to the operating system required to support Data Tethers. We describe the dynamic code rewriting process and give benchmarks of micro applications that illustrate the performance of individual pieces of the system, such as the cost of running rewritten applications. We also provide benchmarks of real user applications, showing that Data Tethers can work in realistic cases with minimal user-visible overhead. Finally, while the original motivation for this work was to protect data on mobile devices, the technology also suggests many other interesting uses.

1.3 Use Cases

To demonstrate the system's goals, we present examples of system use. First, consider a company that collects customer data, including sensitive information such as Social Security numbers, which should not be leaked. The company labels these fields in the relevant tables with a policy in their "Data Tethers compliant" database. When employee **x** queries the database and retrieves customer data, it is sent in the Data Tethers network format. Employee **x**'s Data Tethers laptop recognizes this, checks whether the machine is in a valid state for the policy, and

labels the data before passing it to the user's application. The instrumentation process is started on the client application, and the data labels are tracked as the application manipulates the data. When employee **x** saves this data or any derived data to his hard drive or flash drive, or sends the data, accidentally or intentionally, over the network, the data is automatically encrypted and the appropriate policy is attached.

As another example, user **x** has written a document that he wants to give to a friend to proofread, but does not want forwarded to other people or leaked onto the Internet. Currently, even though the user may trust his friend to some degree, he has no control over his document after it leaves his computer. The friend may have poor computer security and be vulnerable to Trojan horses or viruses, or his misconfigured peer-to-peer client may be sharing everything on his computer. With Data Tethers, the document owner can specify that the document can only be read by his friend, that it can only be read for a specific period of time, or that his friend must be running an up-to-date virus scanner. It also guarantees that with the proper policy, even if the data is accidentally leaked by the friend's poor choice of peer-to-peer configuration, it will not be readable by downloaders.

A third example is the control of personal information. Users are often required to give websites personal information such as credit card numbers, Social Security numbers, or addresses. Details on who this information is redistributed to, how long it is retained, how it is used, etc., depend on company policy, which is often not easy to interpret, and may change at any time. With Data Tethers, the user can constrain how his data is used. For example, he may attach a policy to their credit card number specifying that it is only accessible for one week from the current date, but his contact information may be retained indefinitely. Of

course, the user will have to trust that the company will not attempt to bypass its own Data Tethers system maliciously; however this is a reasonable assumption, since honoring its commitment to use Data Tethers is in the company's interest.

1.4 Roadmap

The rest of this dissertation presents the design and implementation of the Data Tethers system. Chapter 2 is a comparison of Data Tethers with full disk encryption and other technologies currently available to solve the data loss problem. Chapter 3 discusses the design of the system as a whole. In Chapter 4 we discuss information flow tracking, which is used for policy propagation. Some example environmental policy monitors are outlined in Chapter 5. Chapter 6 details the changes to the operating system that were required for Data Tethers. The mechanism we use for dynamic recompilation is outlined in Chapter 7. The policy language and the policy database design is laid out in Chapter 8. Details for the policy monitor and policy server are discussed in Chapter 9. We then present performance data for various system components in Chapter 10. Chapter 11 is related work, including discussion of the many flow tracking systems that have been implemented. In Chapter 11 we outline future directions for the Data Tethers system, or its derivatives. And finally in Chapter 12 we conclude and summarize.

CHAPTER 2

Current Solutions

Data loss is a problem that has existed since the invention of portable media, and that has occurred in a wide range of settings, from government labs losing nuclear data from unwiped hard disks, to lost flash drives containing valuable personal information. Despite the prevalence of the problem, good solutions have been elusive. Technological solutions such as full disk encryption, partial disk encryption or leakage detection by data stream scanning have failed to plug the data leakage hole. Similarly, organizational policies and awareness training have similarly met with poor results.

Data leakage protection can be organized into three main categories, depending on the state of the data when protection is required. Data-at-rest systems protect data that is persisted in some type of storage, and mainly includes some type of disk encryption. Data-in-use is data that is actively being used, generally in memory. Data-in-motion is data that is being transmitted from place to place over the network. Protection of data-in-use and data-in-motion is generally done by scanning or leakage protection software, either installed on the endpoint (end user machine) or as part of some network appliance. Data-in-motion can also be protected by an encrypted network connection, for example by VPN or SSL, but this only protects from third party eavesdropping, since this still allows the data

to exit the machine via the network. Of course, information security policies apply to data in all three states, and in some sense the all data protection solutions can be viewed as tools to enforce sound information security policies.

In this section, we will examine current options to protect against data loss, and their vulnerabilities. These solutions include full or partial disk encryption, scanning or data leakage detection software, and information security policies. The focus of this section will be on *commercial* solutions, as opposed to research based solutions, which will be covered in the chapter on related work.

2.1 Partial or Full Disk Encryption

Partial disk encryption, sometimes called file system-level encryption, is encryption of files at the individual, folder, partition, or file system level. These products offer many advantages:

- the ability to install them on already running systems
- reduced overhead for files that are not sensitive, such as operating system related files
- the ability to use unique keys at the file granularity
- the ability to do incremental backups of only changed files, rather than an entire encrypted volume

Despite these advantages, partial disk encryption has one major flaw: there is no enforcement that copies of the encrypted data must also be encrypted. There are many cases where copies of data are made. In some cases, a user chooses to

copy data, and may or may not remember to encrypt the new copy. In other cases, programs make temporary copies of files while they are being edited or processed, copies which are not encrypted, or the operating system itself may swap out pages of sensitive data to disk. In addition to not being encrypted originally, even when deleted, the data contained in these files continues to reside on the disk.

Given the nature of the problems that partial disk encryption has, one natural solution to the problem would be to simply encrypt the entire disk. This would mean all copies of the data written to disk are encrypted, solving the major flaw of partial disk encryption. However, full disk encryption has several major drawbacks as compared with partial disk encryption.

- difficult to install on already running systems
- encryption overhead is incurred for all files, though this varies based on implementation
- few keys, sometimes one per disk, sometimes one per sector
- entire volume must be backed up or special encryption-aware backup software must be used
- all-or-nothing protection, allowing anyone with legitimate access to bypass the protection it offers

Because of these trade-offs between ease of use and security, many products offer a choice of encrypting at the file, folder, partition, or disk level.

There are two categories of disk encryption, software-based and hardware-based Software encryption is offered as either part of the operating system or as

add-on programs. Hardware encryption can be built into the hard drive itself or as part of the disk controller. Within these two categories there are many variations in implementation, such as number of keys, key storage techniques (e.g. encrypted on disk or stored in a Trusted Platform Module(TPM)), whether or not they use two-factor authentication, and details in how the boot sequence is handled. While it is impossible to exhaustively list all products available and all possible features, we will present the most dominant players in the current market.

2.1.1 Software based encryption

The modern versions of the popular Microsoft Windows operating system ship with two disk encryption options. The first of these is the Encrypting File System(EFS). EFS allows the user to encrypt at the file, directory, or drive level, though it effectively encrypts at the file level, since specifying that a directory or drive is encrypted results in each file contained within being encrypted. This means that directory structures, file names, and other metadata are visible to any user, unless restricted by NTFS permissions, which of course offer no protection in the event of theft.

EFS has very limited options with respect to key storage. Each file is encrypted with a symmetric key algorithm (AES by default). The key for a file is encrypted by a public key belonging to the user who owns the file, then stored with the metadata for the file. Each user has one public/private key pair for this, which is stored on disk, encrypted with the user's password. The private key may also be backed up to an Active Directory server or a recovery disk to enable file

recovery if the password is forgotten. Encryption and decryption are transparent to the user, once they are logged in[8].

There are several significant flaws with EFS. The most obvious is that it depends completely on the strength of the user's password. If the password is weak, or easily obtained, then all files can easily be decrypted. If the device is stolen while running, the private key can be obtained from memory using various physical memory attacks, such as the well known 1394 Firewire attack or the cold boot attack. Additionally, there is no method to allow the EFS system to determine if the operating system has been tampered with, or if a key logger has been installed, enabling two-stage attacks where the attacker first installs a key logger or compromises the operating system, then later steals the device after obtaining the password. Copies of files to unencrypted drives or folders or over the network are not encrypted, meaning that copies of protected files may be stolen from removable drives, accidental network shares, e-mail, or peer-to-peer network sharing. In fact when files are first encrypted by EFS, the plaintext copies are not securely deleted, but instead merely marked as unused blocks on the file system, allowing for easy recovery by a variety of tools.

The second option offered by Microsoft Windows is BitLocker Drive Encryption, a product which encrypts whole logical volumes, rather than individual files, using AES encryption. Unlike EFS, which works on a per user basis, BitLocker protection is for an entire volume, which may be shared between multiple users, and offers no protection from valid users of the machine. If desired, it can be used in conjunction with EFS to provide security in cases where users do not with other users of the machine to see their files, and do not feel that NTFS file permissions are sufficient.

The BitLocker product is more targeted towards data loss prevention, and has several modes that can chosen, depending on the desired security level. Transparent operation mode uses a TPM to ensure that the operating system has not been tampered with. The disk encryption keys are stored in the TPM and only released to the OS loader if the BIOS and boot loader can be verified to be untampered with. User authentication mode requires that the user enter a PIN or password before booting, and USB key mode requires that a special USB key be inserted before the OS can be booted. Any combination of these three authentication methods can be used, all with options to store recovery keys and certificates in escrow[4].

Depending on how it is configured, BitLocker can be vulnerable to several different attacks. Use of the transparent mode protects against bootkits and rootkits, but because the drive is protected with a single key for all users, if *any* user password is weak, then files for all users are compromised. Also, because it can be booted without authentication, it is vulnerable to cold boot, or any other physical memory attacks. Use of user authentication or USB key authentication modes are vulnerable to bootkits, which replace the usual bootloader with software that logs passwords or certificates. The combination of all three is the most secure mode, but still requires good passwords, as well as security of the USB key, which in many cases could be stored with the computer itself for easy use. This method is also particularly onerous for the user, since it requires pre-boot passwords as well as a USB key to be inserted. Even in this extra secure mode, if a device is stolen while running, or while in the sleep state, it is susceptible to cold boot or physical memory attacks.

For OS **x** users, Apple offers a full disk encryption technology called FileVault

2. FileVault 2 encrypts entire volumes including the boot volume. Encryption keys are protected by password, and stored on a non-encrypted recovery drive. The machine administrator can specify which accounts have permission to decrypt the volumes. At boot time, the boot loader requires that a permitted user log in before the boot volume can be decrypted and the machine booted. Once the machine is booted, any user may log into it and use it, whether they have decrypt permission or not. Recovery keys are handled in two possible ways. The first is that the user is given a 24 character recovery key, that they can use to decrypt their volume in the event that they forget their password. The second is an option to "Store your recovery key with Apple", where it is protected with three personal challenge-response questions[7].

FileVault 2 can be attacked in a number of ways. Because it does not utilize a trusted boot mechanism, it is vulnerable to bootkits. It is also vulnerable to weak passwords, for any user with permission to decrypt the disk. Once running, any user account on the machine can be used to access the encrypted volume, in addition to cold boot or physical memory attacks. With FileVault 2 the recovery mechanism offers additional opportunities for attack, since the recovery key is written down on paper or the answers to the security questions Apple uses to determine whether or not to release a key can often be easily guessed or discovered.

In addition to solutions bundled with the operating system, there are dozens of commercial products available to provide software disk encryption, all with various capabilities. Since the number of these solutions is far too numerous to completely enumerate, one representative product will be discussed, McAfee Endpoint Encryption(EE). This is one of the most complete and mature products

available, and therefore gives a good overview of the types of functionality that can be provided.

McAfee Endpoint Encryption is a full disk encryption product. Where EE differentiates itself is in its flexibility of usage and the central management capabilities it offers. Similar to BitLocker and FileVault 2, it supports preboot authentication if desired, including the ability to securely boot using a TPM. As part of this preboot authentication, EE supports a wide range of single or two-factor authentication devices, including USB keys, smart cards, and security tokens such as the RSA SecurID authenticator, which provides time-synchronous, one-time passwords. Disk encryption can be done using software AES, or using an OPAL compliant encrypted hard drive (discussed further in 2.1.2.) AES keys are kept encrypted via asymmetric encryption, with the private key either stored in the TPM, or encrypted with the user's passphrase on the preboot file system. Recovery can be done by an administrator, or via a self-help recovery system. Devices can also be set to auto boot, booting without authentication, either a single time, or during a specific time frame, to allow for centralized patching or updating of software.

One major difference in EE and operating system bundled FDE software is the ability to specify centrally administered policies at the user, group, or device level. These policies control a wide variety of factors affecting the security of the system. Most policies involve encryption related issues such as type of two-factor authentication required, password strength, logging requirements, whether to use hardware or software encryption, when an OPAL compliant hard drive is available, lists of users who can boot a particular device, recovery options, etc. However, specific times when the user is allowed to operate the machine may also

be specified, and whether a user is allowed to write to removable media can also be controlled[5].

Because of the flexibility of the McAfee EE software, it is difficult to evaluate how secure or vulnerable it is. Configured with optimal settings, it would be safe from most attacks, short of cold boot or physical memory attacks. However, the variety of configurations and authentication mechanisms it supports leaves it possibly open to many different attacks. For example, the RSA SecureID used by many companies in conjunction with the EE encryption software was compromised when hackers broke into the RSA corporate servers and stole data that allowed them to produce the one time password sequences the devices generate[6]. The complexity of the software involved also provides a very large attack surface. If the machine is configured to auto boot prior to being stolen, for example, this information must be visible to thieves, since the software must be able to read this information before securely booting. This would allow thieves to bypass the secure boot system during the specified maintenance window.

2.1.2 Hardware based encryption

Hardware full disk encryption comes in three basic varieties. The predominant type in use today is the self-encrypting hard drive, in which the encryption hardware is built into the disk itself. The second type is the enclosed hard disk, where a tamper resistant case containing encryption hardware is built around the hard disk or disks, depending on the usage. These are often external USB connected drives. The third is where encryption hardware is built into the disk controller or chipset. This approach was originally championed by Intel, and announced as

part of their Danbury chipsets, but was eventually scrapped in favor of hardware AES support in the current chipsets[1].

One major advantage of hardware based encryption is that the speed of encryption/decryption is much faster than software based methods. With normal hard drives, hardware cryptography can easily keep up with the read/write rate of the drive. For SSD drives, some performance decrease is noticeable. This is particularly true for drives using the SandForce flash memory controller, which relies on compression to enhance read/write performance. Since encrypted data cannot be compressed, the performance of these drives degrades considerably. Some of the performance advantage of hardware encryption can be offset by software based systems that utilize the AES encryption support built into current Intel chipsets, as mentioned above.

The Trusted Computing Group Opal specification has emerged as the current industry standard for self-encrypting drives. All major disk vendors currently offer Opal compliant hard drives and SSD drives. The Opal specification allows for ranges of blocks to be specified, which will all share the same AES encryption key. Keys are randomly created by the drive itself and never leave the drive. When the drive is deployed, the internal key is encrypted with an authorization key, which is then used to control access to the drive. The authorization key can be stored in a TPM, or with any other method discussed for software disk encryption. Similarly, recovery methods are not specified, but rely on recovery of the authorization key[11].

Self-encrypting drives offer one key advantage over software encryption: encryption keys are not stored in memory, and thus cannot be obtained by cold boot or physical memory attacks. While this is a significant advantage, it does

not make them a panacea. The protection offered by the drive is still only as good as the protection of the authorization key, which is often just a password, and is subject to many of the same vulnerabilities discussed with software disk encryption. While in theory the authentication key can be deleted from memory after use, there is no guarantee that it is, or that it is not captured by a keylogger while it is in memory. Additionally, many organizations use single sign-on, so the password used to protect the encrypted disk is the same password users log into Windows, their corporate intranets, etc. with, making it much more likely to be available in the machine's memory, if it is stolen while running. And if the machine is stolen while running, self-encrypted hard drives offer no protection if the machine can be unlocked without restarting, which is easily done via physical memory access attacks. Self-encrypting drives also offer the significant disadvantage that if the internally stored key is lost due to some malfunction, all data on the drive is lost forever, since there is no mechanism to restore or recover it. This single point of failure *requires* that these devices be backed up to ensure data is not lost.

2.2 Leakage detection software

Leakage detection software generally attempts to protect data-in-use and data-in-motion from accidental or intentional leakage. Data-in-motion products, sometimes called network data loss prevention products, are generally network appliances, installed at network egress points, that attempt to scan outgoing traffic for sensitive data. Data-in-use products are called endpoint data loss prevention products, and are installed on servers or end-user systems and attempt to detect

and control information flows both internal and external to the company. These systems can monitor instant messaging, e-mail, peer-to-peer file sharing or any other software with the potential to leak secure information.

Data loss prevention scanning can be broken into two categories. Precise methods involve either identifying specific information that should not be leaked, often called content registration, or inserting identifying information into the data that needs to be protected (e.g. watermarking). This data can be explicitly identified by owners, or can be identified by business rules. Precise methods may result in false negatives, but cannot generate false positives. Imprecise methods involve scanning for keywords, pattern matching, statistical analysis, and other techniques that may yield false positives. Most companies combine both techniques in their products.

While there are many solutions available, only the RSA Data Loss Prevention(RSA DLP) will be considered, as an example of the genre. It includes a network appliance and an endpoint module, and can scan data transfer via e-mail, webmail, instant messaging, social media, FTP, web uploads, smartphones, Microsoft SharePoint, database queries, USB removable media, and network file systems. Additionally, the endpoint module can enforce policies restricting transfer of data to removable media, printing, or distribution via the network. Though details of algorithms used are not given, the product claims to offer protection through both fingerprinting and rule based imprecise scanning. Sensitive data that passes through can be quarantined, denied access, or simply logged, depending on the configuration of relevant policy[10].

While scanning methods work in some cases, they fail for several fairly common scenarios. Encrypted data, data compressed using tools not understood by

the software, data that is obfuscated intentionally or unintentionally, and data in binary formats not recognized by the software cannot be correctly identified by either technique. It is also relatively simple for a user with administrative privileges on the machine to simply disable or uninstall the software. This could be done maliciously, out of convenience, because the overhead of scanning large files makes the software tedious to use, or simply because the software is erroneously blocking data transfer due to a false positive. In some cases, vulnerabilities in the DLP software itself have been shown to not only allow access to secure data, but to point the way to the secure data by examination of the data control policies[65]. Of course, leakage detection software offers no security at all when a device is lost or stolen, since they can be disabled by the thief or simply bypassed by directly accessing the hard drive.

2.3 Information Security Policy

Information security policy is one of the key enforcement tools to prevent data loss. Unfortunately exactly what is secure information policy is difficult to define, not to mention enforce. Today's business landscape has an ever increasing number of potential loss vectors. Ubiquitous laptops, tablets, mobile phones, cheap and spacious removable media, cloud storage, peer-to-peer file sharing, and an increasing trend of employees bringing their own devices to work make it easy for data to leave the traditional corporate network. As governments increase their concern about information privacy, the legal requirements for protecting different kinds of data become more and more complex.

In a white paper on data loss prevention, professional services company Ernst

and Young suggest guidelines for companies that are refining their information security policy[3]. These include:

- Identify and classify data
- Prioritize protecting important data first
- Implement a data management lifecycle
- Do not permit unauthorized devices on your network
- Do not permit copying sensitive data to removable media
- Understand data usage flows and your data loss
- Keep policies up to date and employees trained
- Audit compliance

They then suggest evaluating enforcement approaches, and implementing them from least to most complex, using the business rules developed to drive the implementation. While this is certainly a reasonable approach, flaws in all these protection systems have already been discussed. On top of flaws in enforcement, it is frequently the case that employees simply don't follow the policy, usually out of convenience. As a case in point, in 2012, a year after releasing this white paper on information security policy, 401k account information for 27,000 Regions Financial Corporation employees was lost when an Ernst and Young employee mailed an encrypted USB drive with the decryption code included in the envelope[9].

CHAPTER 3

Design of the Data Tethers System

3.1 Design Goals

Currently available products, while offering some protection, do not do a sufficient job of protecting data on mobile devices. This is because these products do not address the core issue: *in mobile systems we not only want to specify who can access data, but also where the the data can be accessed and under what conditions.* In traditional data security, access control focuses on controlling *who* has access to data. Questions of where and under what conditions are irrelevant because where is the secure server or corporate network, and conditions are carefully controlled by network administrators. Security instead focuses on constructing a secure perimeter to keep would-be thieves out, and secure data in.

In the increasingly mobile environment of today's business world, this model is insufficient. Mobile devices cross in and out of the secure boundary at will, taking data with them. Once across the boundary, they can be lost or stolen, and regularly are. Loss prevention products attempt to mitigate this, but physical access to a device in almost all cases gives the attacker vectors to access the device, and this is true for all the loss prevention products discussed. If the data is accessible on the device when it is lost, then it will most likely be compromised.

What is instead needed is a way to specify under what conditions data should be accessible, and simply make it inaccessible when conditions are not safe.

In Data Tethers, we define inaccessible as encrypted, with the decryption keys not present on the device. This is a much stronger level of security than can be offered by other security systems, since they store encryption keys locally, and rely on passwords or other authentication methods to control access. Bad passwords, physical memory attacks, authentication devices stored with the mobile device, and other cases mentioned above can allow the thief to bypass these methods and access the protected data. The Data Tethers system will be immune to these attacks, because if configured correctly, it will not rely solely on authentication to control data access. The attacker is free to control any aspect of the system, but with the keys no longer present on the device, there is no way to access the data.

3.1.1 Environmental Access Control

One key question is how do we define exactly what a safe environment is. The answer to this question can vary greatly based on the specific data we are talking about, and is a trade-off between the security we require and the flexibility we need to use the data effectively. For example, secret nuclear codes require the strictest of security, and should only be accessible in the most absolutely safe environment. On the other hand, e-mail from customers may be sensitive and should be controlled, but often must be accessible in relatively unsafe environments, such as airports or coffee shops, in order to communicate with them in a timely manner. However, even in this case we may wish to limit access to com-

pany e-mail to specific staff members, for example sales associates or technical support staff, who regularly need to work off-site.

It is clear that given the many possible types of data in use that to effectively protect data while providing useful access that we need a very flexible system to defining a safe environment for the specific data under consideration. In Data Tethers, we specify safe environments by attaching environmental policies to the data. These policies are written in a flexible policy language which allows us to specify arbitrary conditions on when data can be accessed. These policies will be evaluated on the client machine to determine if the current operating environment is safe or not, because data may be accessed. Because we envision that there may be new types of conditions to be evaluated, depending on the data, we also would like to build an extensible evaluation system that allows for the addition of new environmental conditions, not deployed with the original system.

3.1.2 Ease of Use

Another issue with existing systems is that user takes insecure action, by accident, from convenience, or to avoid onerous requirements. For example, the user may copy files to an unencrypted flash drive because it is a convenient way to transfer large files between computers. Or he may store his 2-factor authentication token with the laptop because it is required in order for him to boot his laptop. With Data Tethers, we would like to make the system as transparent as possible to the user, while ensuring that data cannot be unintentionally or easily used in an unsafe manner. The system should require no special attention from the user, but should ensure that when data leaves process memory, it is stored in an encrypted

format 100% of the time, whether it is persisted to disk, or sent over the network.

3.1.3 Fine-grained Access Control

The design of the Data Tethers encryption scheme is driven by several factors. Given that we have different policies for different types of data, it is clear that a simple solution like encrypting the whole drive with one encryption key is insufficient. In order to control maintain our goal of not having keys available in insecure conditions, we need to have one key per policy. It is also the case that certain files, such as common operating system or application components, do not require any type of encryption or policy at all. Furthermore, it may be the case that in a particular file, only certain parts of the file are secure and should be inaccessible. For example, a spreadsheet containing customer data may have columns with different accessibility requirements, depending on the sensitivity of the data. Social Security numbers might be very sensitive, while names or transaction records less so. Forcing the user to meet the most stringent requirements to access any part of the spreadsheet significantly reduces the usability of the system. We instead would like to give the user access to as much data as possible, given the current environment.

To address these issues, we propose the following scheme. Data controlled by a particular policy is encrypted with a key specific to that policy. Data is labeled at the byte or word level, allowing the user to see parts of the file that are accessible in the current security conditions, while other parts remain encrypted, though in practice this may be limited by the structure of the data under consideration. Only files that have a policy are encrypted, to minimize overhead and to maximize

usability.

3.1.4 Tracking Policy-controlled Data

One problem with our fine-grained access control model is that data is not simply read-only in general use. Users may also derive new data from old, copy data from place to place, or even combine data with different policies. Deciding how to label this derivative data is a major problem to be solved. The simplest solution would be to "and" together the policies of all data read into the application, and apply this combined policy to all data that leaves the application. Unfortunately this naive approach would have serious repercussions for the system. Very quickly, configuration files, caches, and other non-secure file would become policy-controlled, rendering the applications that require them useless outside the most stringent environmental requirements. It might even be the case that due to changing conditions during the execution of program, that the combined policy could include contradictory requirements that result in all data being inaccessible. Furthermore, many systems files might become tainted, for example virus scanner log files, or even the system log file itself. A solution for this might be to implement a system to declassify these types of files, possibly using some type of scanning, similar to the leakage prevention software mentioned earlier, but these techniques are very inexact, and give us no way to say definitively if sensitive data is in the these files or not.

The core problem is that when data is written out by an application, we do not know where that data came from. One way to determine this is by using data flow tracking. With data flow tracking, the application tracks data provenance

as data is manipulated inside the program. When data leaves the application, we can see from what sources it was derived, and attach the appropriate combined policy. While there are several ways to implement information flow tracking, with Data Tethers we wish to build a system with broad applicability to existing systems. For this reason, we have chosen to augment existing applications on demand, adding code at run time to track data provenance.

3.2 Threat Model

For secure systems, it is critical to understand the exact threat model the system is designed to protect against. The Data Tethers system is design to protect data in the event that a portable device is stolen by external attackers. We assume that the attacker has full access and control of the device, but cannot replicate or spoof the environment the device was designed to run in. Thus, it is not designed to protect against attackers internal to the organization who could conceivably operate the device in an acceptable environment.

This may seem like a strong assumption, but in comparison with existing solutions it is much weaker, since at the very least a safe environment will include a valid user logged into the system, which is exactly the same requirement that full disk encryption and other commercial products require, with additional environmental checks offering additional security. Furthermore, while both Data Tethers and FDE solutions keep all sensitive data encrypted on disk when the device is powered off, Data Tethers does not store keys on the machine, or keep keys in memory while the machine hibernates or is suspended to disk. Thus Data Tethers is at least as secure as existing solutions and in most cases more secure.

Data Tethers is designed as a loss protection mechanism, not as a user based access control mechanism, though policies related to required users or groups are possible. In general though, we rely on existing access control to provide protection from unauthorized access by authorized users of the system.

We also assume that rightful users are non-malicious, and are not attempting to exfiltrate data or bypass the system. While a system that can protect against data exfiltration is definitely of interest, dynamic information flow tracking (DIFT) systems are, at best, always subject to covert channels, even such trivial ones as taking photos of the screen, and in the absence of special languages, cannot be made completely secure from leaks. Thus it is impossible for us to guarantee that data labels cannon be removed by a malicious but valid user who has access to a safe environment. Instead, the role of the Data Tethers DIFT system is to provide a mechanism to keep track of secure data in the system for clean up and correct labeling when persisting, and is designed to operate *only in a safe environment.* When the environment is unsafe, the policy enforcement mechanism should remove access to the data, and the DIFT system will never be used.

For similar reasons, we do not consider the case where the attacker steals the device, modifies it, and returns it to the owner. While we do verify that the operating system has not been tampered with on boot, regular user applications may be rewritten to leak information once the machine is back inside the safe operating environment. In this situation, the user should clean and reinstall the software on potentially compromised machines, or the authenticity of the application and its associated libraries should be verified.

Another assumption we make is that the policy attached to the data is suf-

ficient to detect that the current environment is safe in a timely manner. While secure data is always written to disk encrypted, it can reside unencrypted in memory when the environment is safe. While Data Tethers cleans up secure data extremely quickly when an unsafe environment is detected, the transition from safe to unsafe cannot be detected instantaneously. This transition interval is a window of attack, particularly for physical attacks such as cold boot. Proper selection of policy to minimize this window is essential. While this is a weakness of the system, environmental policies are centrally managed and applied to data where, at least in principle, experts in the area can design secure policies. Other security mechanisms, like passwords, are also considered essential but, unlike environmental policies are individually chosen by users, .

3.3 Operating System Assumptions

We make several assumptions about the operating system in Data Tethers. First, we assume that the operating system itself is trustworthy. Encryption keys for policy-controlled data are stored in kernel memory; as a result these keys are at risk if the OS has been tampered with, or a kernel debugger can be run. To ensure that the OS itself has not been compromised, we could use a trusted platform module (TPM), a hardware device available for newer motherboards that certifies that the BIOS, boot sector, and OS have not been tampered with and provides direct attestation of their correctness to remote hosts [2]. The policy server would send keys only when the remote system attests that it is in a secure state. TPM-based systems are vulnerable to cold boot attacks [34] or specialized hardware that can read memory directly, but we are unaware of any defenses against these

attacks with current hardware. Data Tethers is itself, possibly the best defense against these types of attacks, since in general, if the system works correctly, in an environment where cold boot is possible, the encryption keys will have been destroyed and will no longer be in memory.

We also assume that we can reliably determine environmental conditions like location, presence of a particular RFID tag, etc. We do not claim to handle covert channel attacks, attacks utilizing program counter manipulation, reading from processor cache memory or out-of-band data disclosure. We could, in theory, detect some covert channel attacks, like connecting to remote port numbers derived from data, but the current system does not do so. Naturally, we are also unable to prevent photographs being taken of the screen, recording of sound from speakers, or other leakage of data through physical means, though these are not major sources of data leakage in current systems.

3.4 Platform

Our target platform is primarily single-user machines, particularly laptops and other portable devices which periodically leave the secure office environment. While data is always tethered, whether in a database, a file server, or on a local machine, the overhead of a full Data Tethers implementation on the server side may be prohibitive, so we envision that special trusted implementations of server software, such as databases or web servers, will be written to handle policy-controlled data in a more efficient manner than the Data Tethers client software. For example, a secure FTP server may leave files in encrypted form, simply passing them along to the client without unpacking them, while still insuring the

policies are properly maintained.

Data Tethers limits encryption and special handling to tethered data only, minimizing its performance impact. Operating system files, shared libraries, executables, and other files containing non-user data generally do not have policies attached, though it is not precluded for special cases where this is desirable. For example, executables derived from policy-controlled source files would naturally also be policy-controlled, or we may want to control access to a sensitive prototype application.

3.5 OpenSolaris

One of our goals was to demonstrate integration with modern operating systems. We also wanted to demonstrate that the performance was acceptable for typical user applications like word processors and spreadsheets, something we could not do with an operating system written from scratch. We chose to work with OpenSolaris, an open source version of the Solaris operating system, for two key reasons. First, targeting the SPARC processor with its small RISC instruction set makes instrumentation much simpler. Second, we wanted to use the Dyninst library, which for the SPARC architecture only supports OpenSolaris. However, everything in our implementation is extensible to the x86 architecture, and other operating systems such as Linux, at the cost of a more complex implementation due to the larger number of instructions and addressing modes available in x86.

While there are no currently available SPARC based mobile devices, we feel this is not relevant, since this is a proof of concept system, rather than a commercial piece of software. The goal of the system is to prove that is it feasible from a

performance perspective, and that it works correctly. In this situation ease and speed of implementation are more important than a practical demonstration on an actual mobile device.

3.6 Attaching Policies to Data

Policies are attached to data in three ways, depending on the data's state. First, we prepend files with a unique 256-bit marker followed by one or more policy IDs for the data contained in the file. For network streams, policy-controlled segments begin with a unique 256-bit marker, followed by a start tag that includes one or more policy IDs, followed by data in encrypted form, and closed with an end tag. This format can be wasteful, and in the worst case could have one segment per word, increasing the file or stream size significantly. Third, data in user space memory is labeled at the word level, with one word of label per data word. Each bit of the label indicates the presence/absence of a particular policy, which limits the number of policies per process; but for most cases this is sufficient and is similar to previous work [22, 40, 52, 59, 66]. Labels are stored in the user process's address space, so no switch to privileged mode is required to propagate labels. While in the current prototype these labels could be modified by the application, certain SPARC processors allow us to ensure that normal user code cannot tamper with this data. This will be discussed in the implementation section (Section 6).

In the prototype implementation, policies are added to files manually. In a full implementation, tags could be added to data in several ways. Perhaps the most common way would be via organizational data security policy, where

all documents that belong to the organization have a default policy attached, or a policy based on the project or group they are associated with. Interactive labeling of ranges of data or files by the owner would be another method. Labeling specific tables or columns of a database, e.g. credit card numbers, is another way to attach policies.

3.7 Policy Propagation

Policy labels must be propagated whenever the data is copied. The dominant previous approaches are specialized languages, specialized hardware, and dynamic code rewriting. While recent research has focused on specialized languages or hardware due to the perceived high cost of dynamic recompilation, a primary goal of Data Tethers was to demonstrate that this approach was practical in a real computing environment. Thus, we could not rely on special hardware or expect that every application be rewritten and proven correct, given the wide range of user applications available; nor was limiting the user to secure applications desirable.

With the dynamic rewriting approach, we attach to a running process and add instructions into the executable, for example, augmenting load, store, and arithmetic operators with instructions to copy or combine labels. Rather than creating our own code to perform dynamic rewriting, we use the Dyninst library [36], a joint project developed at the University of Wisconsin and the University of Maryland. This library provides functionality such as parsing of instructions in memory, code snippet creation, management, and insertion, and register liveness analysis that are the basic building blocks for dynamic recompilation.

While dynamic recompilation can be costly, we validated two assumptions that make the approach feasible. First, our target platform is personal computers and user applications. Most of these applications use small percentage of a modern computer's processor time. Multiplying this small percentage by a factor of four, five, or even ten will not result in a noticeable degradation of the system. Second, for many of these applications, much of the code never touches protected data, but instead deals with rendering menus, buttons, animations, etc. By carefully identifying portions of the code that manipulate the data of interest, we can limit the impact of the dynamic recompilation process. There are few references for exactly how much processor time the average user application takes, or what percentage of instructions operate on actual user data, but we will show that for several representative applications, our assumptions hold.

Policy data must be stored in memory, preferably in user space to avoid switching between user and kernel mode. We store labels at the word level, with one label word for every word of controlled data. These labels are stored in "shadow pages" in user space memory. A new segment is added to the process's address space to hold these pages, and the address of the shadow page is calculated as a constant offset from the data address, modulo the maximum integer that will fit in the word (Figure 3.1). The modulus is implemented as a simple wrap around (i.e., stack addresses roll over and map into the bottom of the shadow segment) which limits the size of the stack, but this is sufficient for a prototype. In adding a segment, we do not double the memory requirements for the application. The segment merely contains data structures necessary to support these pages. Shadow pages are only allocated for pages that actually contain labeled data.

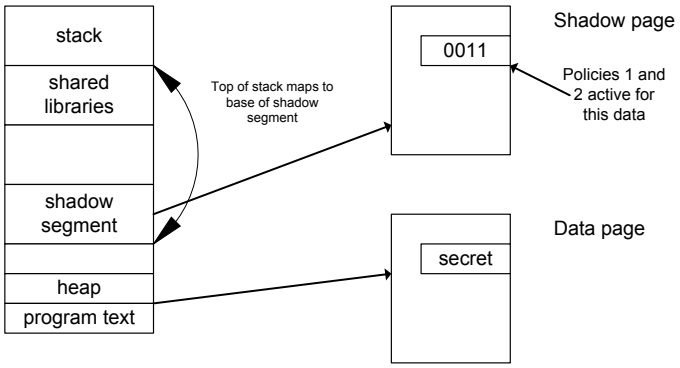

Figure 3.1: Simplified shadow memory layout for a process.

Bad or malicious pointer arithmetic could result in user code that intentionally or accidentally modifies labels, but alternate architecture-dependent implementations with the same computational complexity are secure. For example, the MMU of the SPARC processor allows a process to associate a second address space, which could be used to store labels. Currently, this feature is only used by the kernel to quickly write data from kernel to user space, but it could be added to any process. Special instructions are used to write to this secondary address space– instructions which cannot currently be used by applications since the context switching code does not program this secondary address space register in user mode. By pre-scanning an application to ensure that the original program does not use these instructions and prohibiting self-modifying code, we could use these instructions to store tamper-proof labels. While this is a superior implementation, it requires substantial changes to core kernel code, would have little effect on performance measurements, and so has not yet been implemented.

3.8 The Data Barrier Concept

Policy-controlled data in the Data Tethers system exists either in an encrypted, packaged form or unencrypted and labeled in process memory. Conceptually, we create a *data barrier* around a process, with any data crossing this barrier being converted from one form to the other. This barrier is implemented in the kernel. Because various devices use different interfaces in the Unix kernel, we implement the data barrier in different ways for different devices. For disks, we integrate at the VFS level, allowing us to be agnostic to the file system. For the network, we augment the SockFS interface, which is just above the transport layer, allowing us to send tethered traffic over any transport. Some types of peripherals are excluded from the barrier for practical reasons, such as the video card or the sound card, though we do allow for policies that restrict writing to these devices and protect them from reads when policy-controlled data has been written to them. We currently restrict some types of operations, such as direct writes to the network card. While conceivable user applications could want to package their own transport layer packets, normal applications rarely do, and it is difficult (or impossible without knowledge of the transport) to encrypt only the data portions without destroying header information. A similar argument can be made for raw disk access.

3.9 Environmental Monitoring

Data Tethers specifies environmental conditions under which data is accessible. These may be security requirements such as the presence or absence of software like virus scanners, some types of user identity verification, location, or almost any

other measurable status. The policy monitor, which performs this environmental monitoring, is a user-space application running at a special security level added to OpenSolaris for Data Tethers. This security level limits its access to the system to functions it requires, but prevents any user, including root, from starting, stopping, controlling, or tampering with the policy monitor. Due to the flexibility of policies, the policy monitor accepts pluggable modules that can be downloaded when a particular policy element needs to be verified. These modules also run in a sandbox and are allowed limited access to the system. Each policy element specifies the granularity of detection it requires, both the length of time between status checks and the length of time a violation may be tolerated. While many policy elements require strict enforcement, others may not, for usability reasons. For example, a policy that requires a wireless network connection may allow for brief outages of that connection. The policy monitor is responsible for both initial verification that the system's state satisfies the policy and for notifying the system when environmental changes invalidate a policy.

When a policy becomes invalid, the OS encrypts in-memory labeled data for that policy for each process containing such data. When suspended to memory or disk, a similar process is undertaken for all policies active in the system.

3.10 Cryptography

Data Tethers uses AES in CBC mode with 256-bit keys for encrypting data. Keys are stored permanently on the remote policy server, and are delivered to the policy monitor only on receipt of a signed certification of the system state, including an attestation that the operating system and Data Tethers components

are uncompromised. Keys are stored only in kernel memory on the Data Tethers machine, never on disk, and are destroyed on notification of a policy violation or on shutdown, suspend, or crash. Data may be subject to multiple policies, in which case it is encrypted sequentially with all applicable policy keys, in numerical order by policy ID.

3.11 Remote Key Storage

As noted before, keys in Data Tethers are never stored locally when the device is in an unsafe environment. Instead, they are stored remotely, in a machine referred to as the policy server. The policy server is fundamentally just a database containing policy IDs, the policies themselves, and the associated decryption keys. The policy server relies on attestations from the Data Tethers clients that the policy components are satisfied. In the event of a policy change or revocation, the policy server can also push notifications to clients.

3.12 System Operation Overview

To understand how the system works, is useful to understand the basic flow of data through the system. The major components are shown in figure 1.1, along with the data flows between them. The tethering system starts when the kernel encounters a Data Tethers start tag on data read in from disk or arriving over the network. It then parses the tag for the policy IDs for the data, and forwards these IDs to the policy monitor. The policy monitor forwards these IDs to the policy server, which returns the policy itself. The policy monitor then verifies that the

current state of the machine meets the policy requirements, and sends a signed attestation of this to the policy server, along with environmental state evidence provided by the monitor, which replies with the encryption key for the policy. The policy monitor forwards this key to the kernel and adds the policy to the list of policies that are currently being monitored. The kernel then decrypts the data with the key, labels it in shadow memory, adds a watch point for the buffer, and passes it along to the user application. It also starts the dynamic recompiler, which waits for application watch point faults to begin adding instrumentation. When the application writes back to disk, the kernel re-encrypts and rewraps the tethered data in the policy tags before writing it to disk. One thing to note is that the remote connection to the policy server generally only happens the first time a policy is encountered, making subsequent accesses to data with the same policy much more efficient.

CHAPTER 4

Information Flow Tracking

4.1 Tracking policy-controlled data

One of the challenges faced in the Data Tethers system is how to track policy-controlled data as it moves through the system and how to properly label policy-controlled data derived from other policy-controlled data when it leaves the process address space. Information flow tracking allows us to do just this by tracking the flow of data through the running application. There are several methods for doing this type of tracking, including special languages where variables can be annotated with security types, special hardware that tracks labels along with data automatically, and addition of code to existing programs to track labels along with data. Because Data Tethers is focused on existing operating systems, and because we need to measure performance with standard user applications, we elected to dynamically instrument existing binaries with label tracking code.

There are two issues that need to be considered in the Data Tethers implementation of information flow tracking. The first is the underlying basis for the instrumentation that we do, and the second is the way that we implement it. In this chapter, we consider the former of these, and leave the latter for Chapter 5.

4.2 Static vs Dynamic Information Flow Tracking

There are two different types of information flow tracking that can be used. The first of which is static information flow tracking. In this type of flow tracking, labels for variables are known when the application is written. A good example of this would be an application designed for use with secure files. Data that comes in from secure files is all labeled as high security, and data from other files, such as configuration files, is labeled with low security. In this case, with a suitable language, we can perform static analysis on the code and determine if information can leak from the program or not. A similar and much studied case is that of data flow tracking for buffer overflow attacks. In this case, all data coming in from the network is considered to be suspect, and the path the data takes through the code can be deduced from static analysis, barring incorrect pointer arithmetic.

The second type of flow tracking is called dynamic information flow tracking. In this case, variable labels are not known before hand, and can change during program execution. Since the data in the Data Tethers system can have many possible labels, or no label at all, and could include data read in from any source, this is the type of information flow tracking we must do. Dynamic information flow tracking also requires combinations of security labels, as data moves through the application, and is generally much more difficult.

4.3 Explicit vs Implicit Flow Tracking

There are two different mechanisms for information to flow inside of an application. The first is explicit flow which includes loads, stores, arithmetic operations, etc., where data is directly combined or assigned from one memory location to another. The second is implicit flow, where information is transferred by the control flow of the program. For example, if we set the value of a variable **a** to either one or zero, depending on the value of a second variable **b**, the value of **b** is never directly copied into **a**, but **a** is effectively set as a side effect of the flow control statement. As discussed in the implicit flow section (Section 4.3.2), it is impossible without conservatively over-labeling to prevent certain types of information leakage via implicit data flow without the use of special languages that can guarantee noninterference; i.e., that we make no assignments to low security variables inside conditional clauses controlled by high security variables [45, 67]. However, for realistic cases and applications, our implementation handles both explicit and implicit flow tracking.

4.3.1 Explicit flow

Explicit flows are the simpler of the two types of data flows we are considering, but there are several special cases that make it more complex than it would appear at first glance. The most appropriate way to describe how labels flow would be to define a security type system for the language under consideration, and then discuss its properties, and prove its correctness. However, in this case, the language we're discussing is a more along the lines of a pseudo-code, rather than the full assembly language, which we will discuss later, so we will discuss

the problem from a motivational perspective, rather than a rigorous one.

The most basic type of explicit flow is a simple assignment of the form:

```
a = b
```

where **b** is a variable (pointer or value), constant, or expression. In this case, we simple set the label of **a** equal to the label of **b** (the label of a variable **x** is denoted label(**x**) from here forward). If **b** is a variable, label(**b**) is just the label of the variable. If **b** is a constant, then label(**b**) = 0, where a zero label denotes uncontrolled data.

In the case that **b** is an expression, we must first evaluate the label of the expression. For expressions, we only consider the binary operation case, and construct more complex cases recursively. For the binary operation **c**$\sim$**d**, we define label(**c**$\sim$**d**) to be the least restrictive label such that it is at least as restrictive as both label(**c**) and label(**d**) individually. Practically speaking, what this means is that combining data with two different labels results in data having a policy that requires the conditions of both the constituent policies be valid. Of course **c** or **d** could also be expressions themselves, which allows us to evaluate the label of any expression required. Order of operation should follow the normal order of evaluation for the expression being considered. For example:

```
(a + b) * 0
```

should evaluate to have **a** 0 label.

Given these basic tools, we'll now consider two special cases. The first is the case of array indexing, or equivalently, pointer arithmetic. Consider an expression of the form:

```
a[b] = c
```

where label(**a**) = label(**c**) = 0, but label(**b**)$\neq$ 0. In this expression, **a**[**b**] is a memory location, and we need to label it appropriately. Initially, it might seem that this is simply the case that label(**a**[**b**]) = label(**c**), but this is not the case. Consider the case there **a**[**x**] = **x**, i.e. we set the value of the xth location to **x** itself. Now, we let **c** = 1, so label(**c**) = 0, and label(**a**) = -1. However, when we make the assignment **a**[**b**] = **c**, the value of **b** can be determined by simply scanning the array and searching for -1. Instead, label(**a**[**b**]) should be label(**b**$\sim$**c**)in this assignment.

The second case to consider is dereferenced assignments such as:

```
*a = c
```

where **a** is a pointer type. While is may not initially be clear why this case would need to be considered, we could easily have preceded this assignment with an assignment using pointer arithmetic such as **a** = **a** + **b**, in which case the assignment to *__a__ has the same effect as the array assignment above. In fact, array assignment is just a special case of pointer dereferencing, where **a** is allowed to be an expression that evaluates to an address type. Thus, label(*__a__) = label(**a**$\sim$**c**) after this assignment.

4.3.2 Implicit flow

Implicit data flow is data flow that results from the control flow of the program. For example in this code:

```
if ( secret )
```

```
   a = 1
else
    a = 0
```

the value of secret is indirectly stored in **a**, based on the control flow of the application. If we relied on strictly on explicit flow tracking, label(**a**) would be 0, since we simply assign a constant value to it. Clearly this would be incorrect. Instead we would expect that label(**a**) = label(**secret**). While this seems like a straightforward modification, consider the following code:

```
a = 0
if ( secret )
   a = 1
```

Again, in the case, we intuitively see that label(**a**) should be label(secret), but what is interesting in this case is that this is the case *whether or not an assignment is made to* ***a*** *at runtime.* For if statements, this may seem, again, trivial, but in the following example, we see this must be extended to loops as well.

```
// set all values in the array b to 1
while ( a != secret2 );
   b[a] = 0
   a++
end
```

In this example, we see that we must extend our implicit flow tracking to loops, as well as if statements. However, the problem in this case is slightly

more complex, since we must identify all possible assignments that could have been made inside the loop, or else we will leak information about the secret. For example, if the value of **secret2** was 7 and we simply label memory locations assigned to in the loop, we can look at the value of **b**[9] and know that the value of **secret2** was *not* 9, without performing any comparison with a high security variable, which is not acceptable.

The question here is can we identify all possible memory locations that can be modified by a loop, for all initial states? The answer, of course, in the most general case we can't, since we would need to know if and when the loop terminates for all inputs, which is undecidable in some cases. Of course, in a finite memory machine this problem *is* decidable, since we can completely enumerate all states, though this is not computationally feasible.

So, given that it is not either theoretically or computationally possible to determine all possible memory locations that may be modified in a loop, we need to consider alternatives. In some cases we can determine that a loop does terminate for all inputs, which would allow us to correctly determine the set of memory locations that need to be labeled. However, this is by necessity limited, since there is no general algorithm for this, and we can only handle a limited set of loops.

Another option is to consider conservative approaches, that might over-label, but guarantee that data will not leak. What approach we take depends to some degree on what assumptions we make. If we assume that we have enforced bounds checking on arrays, we can simply label all elements of the array with the loop control variable label. However, this does not extend to more complex structures like trees or lists. For these complex data structures involving pointers, if we

assume we have enforced type safety for pointers, we can take the more conservative approach of adding the label of the loop control variable to the label of all memory locations of the type of the variables on the left hand side of any assignment inside the loop.

There are other static analysis approaches that might provide a conservative but more accurate approach, but the value of this approach is difficult to determine. Data Tethers targets already compiled applications, so the language we will deal with in the implementation is assembly, with no type of bounds or type enforcement, and with no guarantee of correct execution. This particular case is extremely difficult to deal with, and of course impossible in the most general case.

Difficulty of analysis issues aside, we also need to consider the merit of the approach from a system standpoint. Applications running on the Data Tethers machine are under the complete control of the machine's owner. While undecidable loops may be rare in normal code, a malicious owner can easily write his own application with the intention of leaking data and defeat our best efforts, unless we take the draconian approach of labeling everything that exits the process with the combination label of everything that came into it.

To see that this is the case, consider the following situation, with a somewhat simplified execution model. Assume that we have an infinite memory space, separate in the case from the memory holding instructions to simplify things, with all locations initially set to zero. Now assume we have a secret S that we want to protect, and our application consists of a single loop, that takes S as a parameter and generates a sequence of non-repeating integers, the termination of which is undecidable (e.g. the sequence generated in the generalized Collatz

problem.) At each iteration, we set the corresponding address for the current sequence number to 1, and terminate when the sequence ends. When and if the program terminates, we dump the entire address space to disk, not including S. By examining the which addresses contain a 1, we can either determine S, or at the least derive information about what S could or could not be. Now, the question becomes, in order to hide S, which memory locations should we label as high security? Since the termination of the sequence controlling the loop is undecidable and the sequence in non-repeating (as must be the case in the Collatz problem), this could include **any** memory location for the application, and so we **must** label every location in order to protect S.

Thus we made the decision to instead assume the legitimate user was non-malicious, and make attempt to limit under-labeling due to track implicit flows, and see if in practice we leaked data. To reiterate a point from the threat model section, the information flow tracking code is meant to keep track of data, while it is being used in a secure environment, not to provide data security.

4.4 Covert Channels

In addition to information leaks due to implicit flows, DIFT systems are also vulnerable to *covert channel* attacks. Some examples of covert channel attacks include methods as simple as taking photographs of data displayed on the screen, to as complex as transmitting bits by monitoring spikes in power usage caused by processor load. While some of these channels are made available because we need to use the data in question, others are unavoidable side effects of the operating system and hardware design. When dealing with covert channels, the

best scenario is to identify and quantify the bandwidth of the possible leak. We did not do this as part of Data Tethers, and instead, common to most other work in this area, we simply ignore the problem of covert channels.

CHAPTER 5

Dynamic Instrumentation

5.1 Overview

As mentioned before, the goal of the dynamic instrumentor is to insert code into the executable to track label information along with data flows so that any data that exits the application has the correct policy attached. There are several methods to do this, including writing special applications with security typed languages, statically instrumenting executables, or dynamically instrumenting running executables, just to name a few. Because the overhead incurred by instrumentation is quite high, we have elected to dynamically instrument running executables, with the goal of instrumenting as little as possible by identifying at run time code that uses policy-controlled data. To do this, we use a combination of modifications to the operating system, watchpoints, and the Dyninst dynamic instrumentation library. This chapter will focus on the implementation of the instrumentor, and changes to the the user space process required to support it. Changes to the operating system are discussed in detail in Chapter 6, but in some cases must be mentioned briefly in this chapter to understand the instrumentor's operation.

In Chapter 4 we discuss two types of information flows: implicit and explicit

flows. Our instrumentation tracks both of these types of flow, and while our explicit flow tracking is correct, our implicit flow tracking implementation is, of necessity, a best effort implementation. Despite this, we show experimentally in Chapter 9 that from a practical perspective, this is sufficient. The implementations of these two types of flow tracking are very different, and so they are discussed separately.

5.2 Dyninst

Instrumentation of executables is a difficult task. To aid us in this, we use the Dyninst library. Dyninst is written in C++ and provides a simple interface to create "snippets" (small pieces of code to be inserted into the application), search for and iterate through both functions and individual instructions, and to insert these snippets as needed. It also includes functionality for allocating memory in the application's address space, controlling program execution, catching application signals, and handling dynamically loaded libraries, to name a few of the more useful ones. A Dyninst application consists of two pieces. The first is the instrumentor program, which attaches to the application to be instrumented and contains the logic to handle the insertions. The second piece is a series of dynamically linked libraries that are loaded into the targeted application's address space and can contain pre-made snippets as well as the Dyninst library code to handle the insertions themselves.

The stock version of the Dyninst library used in Data Tethers required several modifications in order to be useful. The first of these was that in an attempt to abstract out the idea of instructions across different architectures, direct access

to instructions was not allowed. Instead, instructions are grouped into classes, such as "reads memory" or "writes memory". This level of information while sufficient in some situations, was not sufficient in general for Data Tethers, since we needed classes of instructions that were not available. To rectify this, we added an interface to allow direct access to the opcode portion of the instruction. This allowed us to construct new classes of instructions, such as "binary operation" or "unary operation", which more closely fit the classes of instrumentation we wished to insert.

Along a similar vein, Dyninst was not designed to give direct access to operands. Instead, it supports the create of class instances representing the target address of instructions, rather the individual registers. The Sparc architecture supports operands of the form base register + offset register * constant for loads and stores in order to easily support arrays. As discussed in Section 4.3.1, in order to label arrays correctly, we need to combine the label for the offset as well as the label for the value register. In addition, to support certain optimizations, it is also necessary to keep track of which registers are in use. Thus we also modified Dyninst to expose the register mnemonics for each instruction. Similarly, Dyninst had no ability to specify which registers were used in the code it generated, limiting our ability to optimize registers used for label tracking. This functionality was also added.

Finally, the last piece of functionality we needed from Dyninst was the ability to determine the current instruction and function being executed (for purposes that will be discussed later). An interface was added to allow this. Additionally, this project used the Dyninst library more extensively than other projects, and many bug fixes were made, some in collaboration with the Dyninst development

team.

5.3 Operation of the Dynamic Instrumentor

In order to understand how the dynamic instrumentor works, we need to first have a high level understanding of its operation: how it is triggered, how it interacts with the process and how it decides which sections of code to instrument. In this section, we will discuss these mechanisms, before digging deeper into inserted code.

When an application accesses policy-controlled data, the instrumentation process is notified by the OS. This process determines in which function the watch point occurs and instruments the entire function. The function is then considered *tethers safe*, and the monitoring process for labeled data access is disable at the start of the function and re-enabled at the end. Calls to other functions within the instrumented function are handled in one of two ways. Monitoring is turned on and off before and after calling a function that has not been instrumented. Calls to instrumented functions are not modified.

We originally instrumented at the block level, but a detailed analysis of which registers were used in following blocks was required in order to determine the full list of blocks to instrument. This analysis was costly and, practically speaking, resulted in most of the function being instrumented in any case. Also, the enabling/disabling the monitoring process involves a system call, which is fairly costly both code-wise, and because of the switch from user to kernel mode. Making these calls at the beginning and end of each block was prohibitively expensive. Additionally, Dyninst inserts a "trampoline" in the place of instrumented code

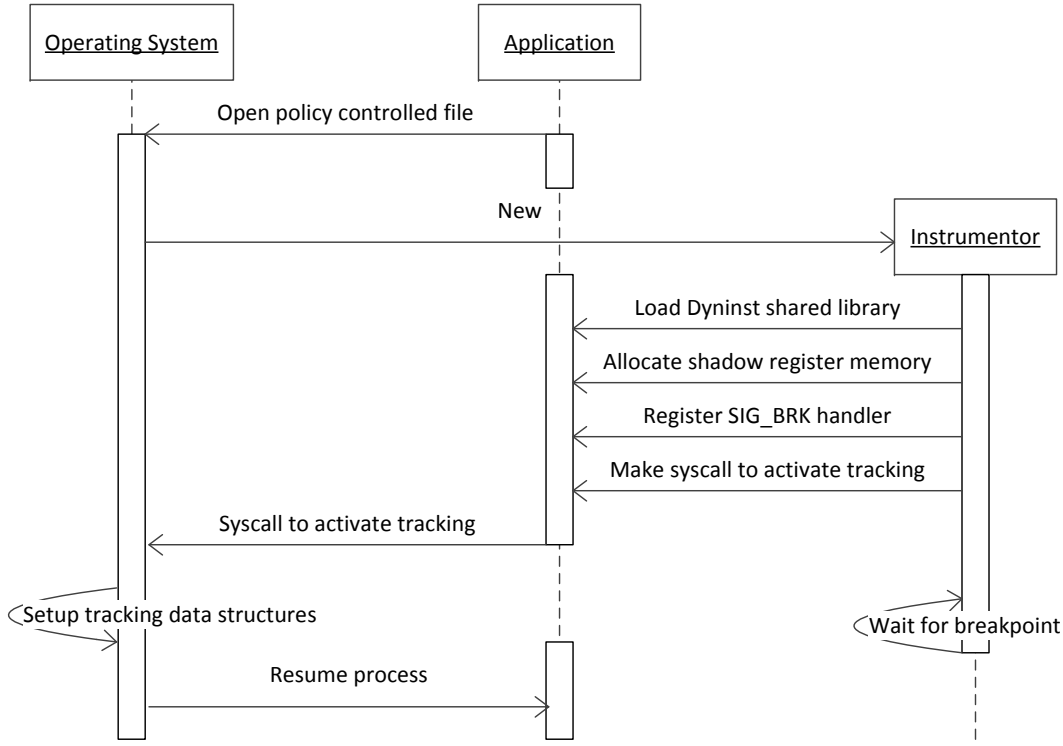

Figure 5.1: Startup for the instrumentation process.

which saves and restores active registers, then jumps to the instrumented code segment. Inserting this code at the block level further increased the cost of instrumentation. Choosing to instrument at the function level resulted in better performance and increased simplicity.

In figure 5.1 we see a sequence diagram for the start-up of the instrumentation process. When an application opens a policy-controlled file, the operating system starts the instrumentation process, passing it the process ID of the application to be instrumented. The instrumentor then attaches to the process address space and loads the Dyninst client shared library. This shared library creates a new thread in the client process which handles IPC requests from the instrumentor, for

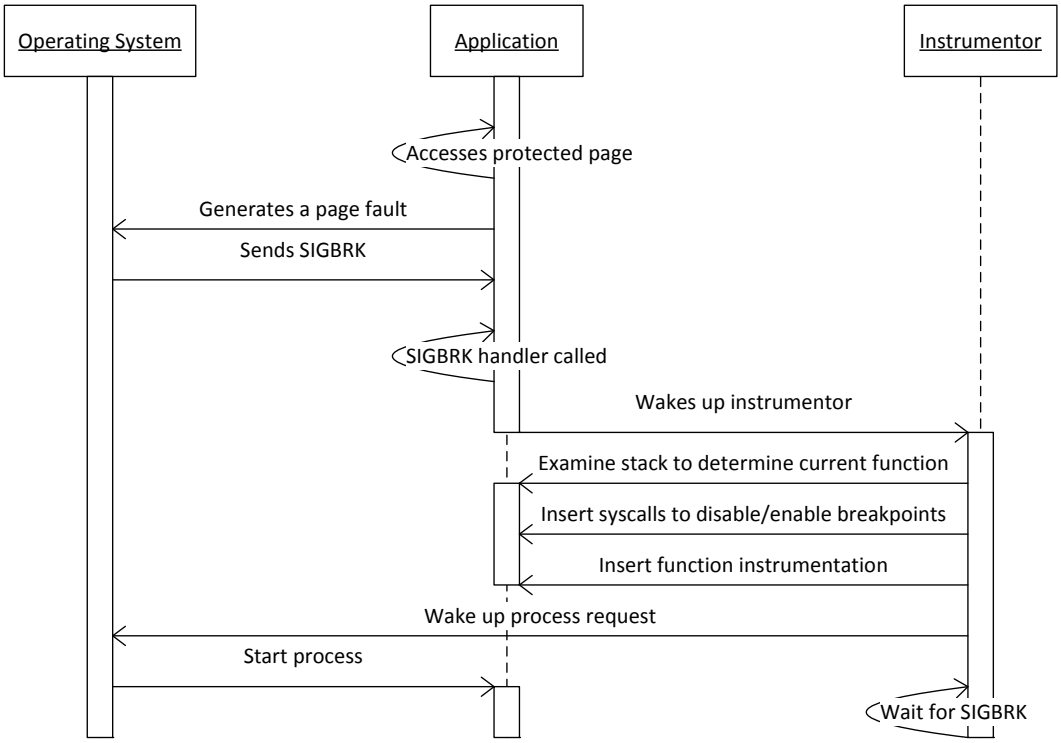

Figure 5.2: Code insertion by the instrumentor.

example, when inserting new code snippets. The instrumentor uses this library to allocate memory in the client address space to hold the register labels and implicit label stack needed by the instrumentation, then registers a handler for the SIGBRK signal, the role of which will be discussed in detail in the next section. Finally, the the instrumentor makes a syscall in the client address space to activate tracking, which initializes tracking data structures in the operating system and allows the process to resume, then sleeps until a SIGBRK occurs.

To see how the instrumentor works in action, consider figure 5.2. As mentioned before, label controlled data is protected by watchpoints. (The implementation of this is discussed in detail in Chapter 6.) When an application instruction

in a previously uninstrumented function attempts to access a page containing labeled data, it generates a page protection fault. The operating system then checks to see if the address itself contains a watchpoint. If it does not, the OS disables page protections and single-steps the process, then renables them. If it is indeed policy-controlled data, the operating system sends a SIGBRK to the process. This SIGBRK is caught by the signal handler registered by the Dyninst client library, which then wakes up the instrumentor.

The instrumentor first needs to determine which function is currently executing. It does this by examining the call stack, then using Dyninst to find the start address of the function that needs to be instrumented. Once the function is located, the instrumentor inserts system calls to disable/enable watchpoints at the start and end of the function, respectively. It then generates the function instrumentation, and inserts it. (The details of this process are discussed in the following two sections.) After instrumenting, the instrumentor makes another system call to wake up the process, and at the same time disable watchpoints, since the current function is now considered "safe". The instrumentor then waits for the next SIGBRK signal.

5.4 Explicit Flows

Dynamic recompilation for explicit flows is straightforward, though complex in detail. For loads or stores, we insert two instructions. For loads we add instructions to copy from the source address's shadow page address to a register, then from the register to the destination register's shadow memory, and similarly for stores. If there are no free registers, more instructions are needed to save and

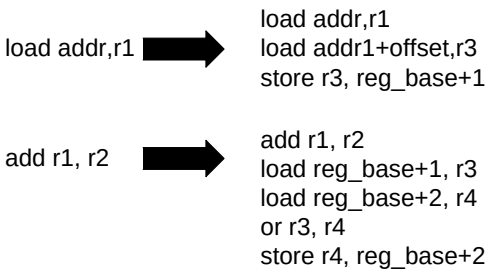

Figure 5.3: Example instrumentation for explicit flow tracking.

restore register contents. If registers are free we can use the free register rather than the shadow store. Information can leak due to pointer dereferencing, so for indirect loads or stores, the label of the pointer itself is ORed with the data label. For arithmetic, bit, or logical operations, we copy the labels from the two shadow memory locations to registers, OR the two values, and store them in the shadow memory for the destination register of the original operation, a total of four instructions. Again, in some cases we must save and restore registers, and in others we can use free registers and save on instrumentation instructions. Figure 5.3 shows examples of both types of instrumentation.

We only instrument instructions that modify actual data. Loads that result in function calls or load constants, for example, are not instrumented. However, since we disable watch points inside of the instrumented function, we instrument every such instruction, whether we are sure that it touches labeled data or not. While we instrument more instructions than absolutely necessary, since we disable watch points inside of the function and it is impossible to analyze which registers might contain labeled data, we must do so to guarantee correctness. We could instrument at a finer granularity, but the cost of multiple system calls, as well as watch point faults, outweighs the benefit gained. Instrumentation of instructions

that do not access labeled data simply results in copying empty labels which, while inefficient, is correct. The instrumentation procedure itself is lightweight and has little impact on application performance, but certain shared library functions tend to be instrumented every time an application using them accesses tethered data. The best examples are the IO functions in the C and C++ standard libraries. Commonly instrumented functions like these can be saved in instrumented form by the Dyninst library, and then reused, rather than having to be reinstrumented.

5.5 Implicit Flows

Implicit flow tracking works in two phases: work done during the instrumentation phase, and program behavior at run time. Consider the example pseudo code below:

```
if ( secret1 )
   // write out 1
else
   // write out 0

while ( a != secret2 );
a++
// write out a
```

This example contains two control flow statements conditioned on policy-controlled data. While the program flow is straightforward in a high-level lan-

guage, it can be quite complex in assembly. The first stage in the instrumentation process determines the control dependence region (CDR) for each branch instruction [29]. The CDR is the set of instructions that execute under the control of the condition and can be readily determined from the application's control flow graph. In our example, the CDR for the *if* conditional includes everything in the *if* and *else* clause, and the CDR for the *while* statement includes all code in the loop. The CDR for a branch instruction includes all functions called within any block in the CDR. All code blocks inside of a CDR share the same security context. Since CDRs may be nested arbitrarily, we use a stack to keep track of the current implicit security context. At the start of a new CDR, we insert instructions to OR the label of the conditional variable with the current implicit security context from the top of the stack and push the new context onto the stack. On exiting a CDR we add code to pop the current context from the stack. Inside a CDR, we must ensure that the current implicit context label is added to any register or memory address that is written. To do so, we OR the implicit context label from the top of the stack with the label of any memory address that is used in a store instruction and the label of any target register of any other operation. In addition to explicit branch instructions, we instrument register indirect jumps using the label of the register containing the jump address.

To see an example of an implicit stack and the corresponding CDR, see figure 5.4. In this example, we have two nested control dependence regions. The first corresponds to the branch instructions B1, and the second with B2. All data used in blocks 2, 4, 5 would include the label of the data used in the compare instruction for B1, and data in block 3 would include the label B1—B2. When the both branches of execution rejoin into one, the current label is popped off the

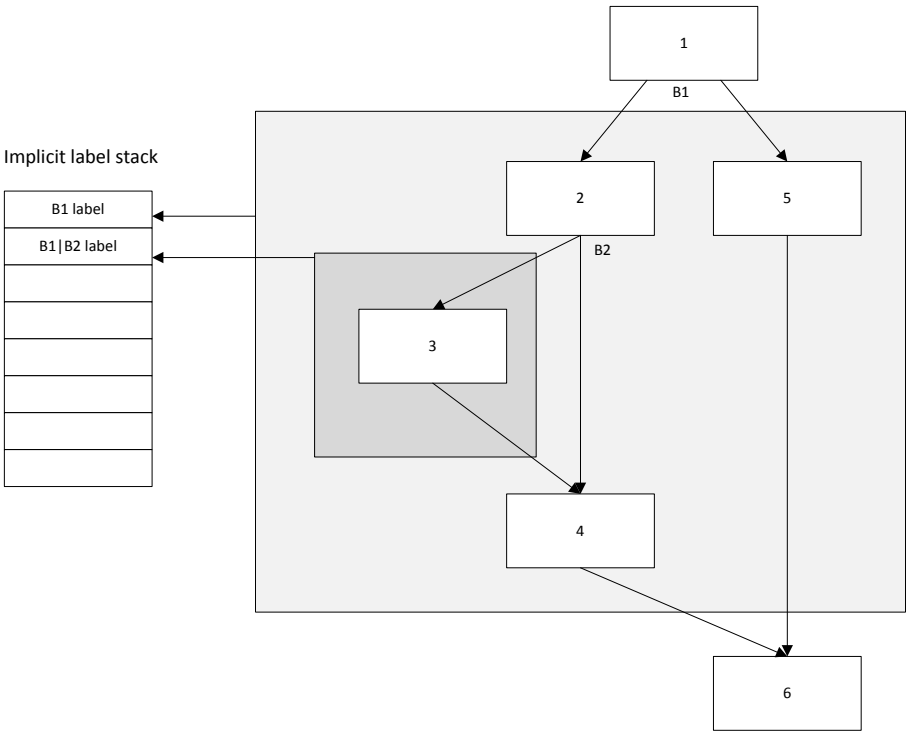

Figure 5.4: CDR graph and corresponding implicit flow stack.

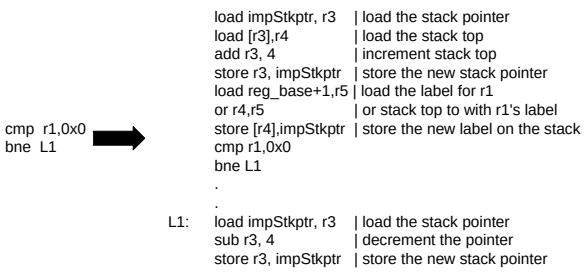

Figure 5.5: Implicit flow instrumentation of a simple *if* statement.

implicit label stack.

Figure 5.5 shows an example of implicit flow instrumentation of a simple *if* statement. We start by loading the current implicit stack pointer and the top of the stack into registers. We then increment the stack pointer, OR the label of the register used in the compare with the current implicit label, and store it on the stack. On exiting the instrumented block we decrement the implicit label stack pointer and continue with execution.

Implicit flow tracking can be costly, but most applications need only a limited amount. Therefore, we make copies of all implicit flow tracking code, and execute it only if the current security context is non-zero. This means the cost of implicit flow tracking is only paid when it is absolutely required. Additionally, we note that we only need to deal with implicit labels in certain situations. We must OR the implicit context with any stores, and we when we pop an implicit context off the stack, we must OR the implicit label we pop with any registers that either have data loaded into them in the previous context or store the result of a binary operation during the previous context.

5.6 Watch Point Costs

While watch points allow us to identify instructions that access labeled data, they have a hidden cost. Some processors have registers for watch points, but they are generally implemented by modifying page protections. When an application accesses the page, a protection fault is generated, and the fault handler looks up the address in a map to see if it should generate a watch point fault or continue execution. Since only a few bytes of a page might be watch pointed, we can generate many spurious page protection faults, severely affecting an application's performance. This is particularly true of the stack, and special care is taken to handle, minimize, and clean up stack watch points in our code. Fortunately, many large-scale applications use slab allocators for heap memory, which minimizes mixed data types on pages, thereby reducing the impact.

CHAPTER 6

Operating System Modifications

The Data Tethers system is tightly integrated with the OpenSolaris operating system. Many changes were made to the OpenSolaris kernel to support the Data Tethers system. These changes included:

- modifications to the file system and network layers, to support detection and decryption of policy-controlled data
- changes to support interaction with the policy monitor (discussed in Chapter 8)
- the addition of several system calls for the instrumentator
- changes to the memory subsystem to support watchpoints on labeled data
- changes to the process data structures to support tracking which processes are actively using various policies and to support the label space

In this chapter we detail these changes. We also discuss other options considered and the rationale behind the implementations we selected. The changes to the operating system are tied closely to the dynamic instrumentation process, and so we will frequently reference Chapter 5.

6.1 Data Barrier

The first operating system change we will discuss is the implementation of the Data Barrier. The Data Barrier in Data Tethers is more of a conceptual construct than a unified piece of code. The idea is that we track all data that leaves a process's address space, looking for data that has a policy label on it. When this data leaves the process, it *must* be encrypted and properly labeled. Since the only way data can leave a process's address space is via system calls to the kernel, this is the natural place to intercept it.

Unfortunately, aside from the system call interface, there is no unified point in the kernel to insert this code. System calls may result in writing to a file system, the network stack, directly to hardware devices, or to other processes via IPC such as pipes or semaphores. It is possible that the we could handle this at the device interface level, but the resulting implementation would be extremely inefficient. Even at the device level, there are several interfaces, depending on the type of device (block, stream, etc.) and this would not handle IPC like named pipes. Additionally data may also indirectly leave the address space of a process via system calls that map in shared pages, which could not be handled at the device level.

In the current Data Tethers system, we implement the Data Barrier as a proof-of-concept in two places: the file system and the network stack. Because one of these file systems are block devices and network connections are streams, these two implementations are very different, mostly for efficiency reasons. In the following sections, we discuss how these are implemented, then discuss possible implementations of other exit points, such as IPC or shared memory.

6.1.1 File system

The most common source of policy-controlled data is the file system. Because of this, we would like to implement the data barrier in as efficient and generic a way as possible, to optimize the most common use case while enabling it across as many different file systems as possible. Thus, we decided to implement the file system data barrier at the VFS layer. By modifying the VFS layer, we can support Data Tethers on virtually any file system available on OpenSolaris. While it is possible to utilize file system specific features to provide slightly more optimal performance (for example extended file attributes in ZFS), we decided that the resulting loss of generality outweighed the relatively small performance gains.

6.1.1.1 File Format

Because we implement at the VFS level, we embed the Data Tethers specific information as part of the data of the file. This makes the process of managing the file slightly more complex. For example, we must account for the policy data embedded in the file when displaying the file size, seeking within the file, or returning information about the current position in the file. In all cases, our goal is that from the user perspective the file looks identical to the same file without any policy data.

Since we store the policy data embedded in the file itself, shared file systems, FTP, or other file transfer methods can be made Data Tethers aware trivially, meaning that in cases where the application did not plan to actively use the data, but merely intended to transfer it, the Data Tethers process could be bypassed

and raw, encrypted data used directly. This would allow files to be transferred without first decrypting and marshaling them into the network protocol (discussed in the next section), and re-encrypting.

When choosing the Data Tethers storage format, we considered several options, including storing the metadata at the front of the file, storing it inline with the data, or storing it at the end of the file. We ultimately chose to store policy data at the end of the file. This greatly simplifies seeks and keeping track of the current file location, as compared to storing it at the start of the file. It also makes adding to the policy section simpler, since we do not need to shift file data to update the policy data.

While storing it inline has the appeal that it would match the network format, giving us a more unified approach, this option would require constant scanning of the data stream while reading. It would also have a significantly higher storage overhead, since we would have large amounts of redundant data compared with storing it in a central location.

The format of Data Tethers files can be seen in Figure 6.1. As mentioned before, we store the data at the end of the file. To mark that this file has policy-controlled data, we use the the pattern 0xDEADBEEF as the last four bytes of the file. Right above this tag, we store the length of the policy data in bytes, so that we can easily find where to start parsing. Next we have a series of Policy tags which correspond to a particular policy combination. This element includes a PolicyID element with a list of the policy ID's associated with this policy, then a list of data regions controlled by that policy, stored as start byte and extent.

One thing to note here is that extents are number of bytes of *data*, not actual

Figure 6.1: Data Tethers filesystem format

length in file. Because we use AES, which is a block cypher, we encrypt in 128-bit blocks. When the extent is not a multiple of 16, it is padded with zeros before being encrypted. This is important because when we report the size of a file, or perform seeks within it, we need to do so based on the size of the file minus this padding.

This file format was designed to minimize the amount of data we have to store, in the average case, where we consider the average case to be that there are not a large number of different policies associated with a file, and that there may be many regions for each policy. A good example of this would be a spreadsheet where a particular column held sensitive data, or a comma-delimited file where a certain field was sensitive. In the worst case, that each byte is controlled by a different policy or combination of policies, this format would be extremely wasteful, with the resulting file being 10 or more times the size of the original file, though we think this case would be unlikely.

6.1.1.2 Handling files

Having specified the file format, we can now discuss how we handle file IO, given the specified format. When a file is first opened or if the files attributes are requested, for example by 'ls', the operating system checks to see if the last byte is the policy-controlled tag. If it is not, no further special action is taken. If it is, the operating system reads the policy section of the file and either calculates the true file size, if the attributes are requested, or builds a table of policy-controlled byte ranges, if a file open operation is requested. As mentioned in Chapter 5, this is also the point where the dynamic instrumentation process is created.

When subsequent read operations are performed on this file, the operating system first looks up the bytes to be read in the policy byte range lookup table, to determine if the read bytes need to be decrypted or not. If they do, the OS requests the decryption key from the Policy Monitor (discussed in section 6.3), decrypts the data, and passes to to the requesting process. It also copies the appropriate label bits into the shadow space for the buffer written to by mapping in the containing page into the kernel address space and copying the labels.

For writes, if the process has associated policies, we check to see if the shadow page for the buffer to be written has a physical shadow page allocated for it by looking in the segment map, a kernel data structure that contains information on physical pages allocated for addresses in a segment. If so, we check the shadow page for labels for the write buffer, encrypt, and update the policy data at the end of the file.

We also must account for the policy data embedded in the file when displaying the file size, seeking within the file, or returning information about the current position in the file. In all cases, from the user perspective the file looks identical to the same file without any policy data. Since label data is hidden from the user, it should not usually be possible for the user to tamper with labels. However, if file label data is modified, no loss of security occurs, since changing policies or ranges will result in decrypting with incorrect keys, or decrypting unencrypted data. We do **not** require network communication on every read or write, or even on every file open. Once a policy has been validated, we assume it is still valid until we are notified otherwise by the policy monitor; and if data with the same policy ID is open, we do not re-request the policy or the encryption key from the policy server on future opens.

6.1.2 Networking

While files have a well defined start and end, network data is fundamentally different. Unlike files, the operating system does not have access to the complete stream of data, only what has been sent so far by the remote process. Because of this, tethered data tags in network streams must be handled differently than file data. Instead of attaching all policy data at the end of the network data, we instead elected to embed policy information inline in the stream of data. We can see this format in Figure 6.1.2.

This is inefficient, since we must then scan all incoming data for these tags. Another option would have been to transmit policy label data in a side channel, which would have been more efficient, but also much more complex. A side channel would have required creating a secondary connection, and ensuring that the two channels were in sync. This most likely would have resulted in delays, as we would have to wait for data from the side channel before passing the normal channel data to the reading process, but would have saved the time spent scanning incoming network buffers for the Data Tethers start tag.

In order to cover as many transport protocols as possible, we made our changes at the SockFS layer, a generic interface for working with network protocols. We added code that scans for Data Tethers labels to the SockFS read and write functions. When labels are found, we encrypt and package the data (for writes) or decrypt and unpackage the data (for reads). Special attention was required for the network layer to handle fragmented Data Tethers bundles, since encryption is done in 128-bit blocks.

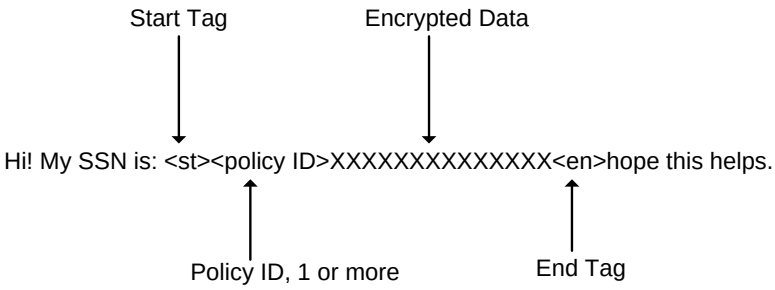

Figure 6.2: Data Tethers network format

6.1.3 Other IPC

The current Data Tethers implementation does not include tracking data transferred between processes by IPC other than sockets, but conceptually this is not difficult and several such systems have been proposed. For example, in the Flume system IPC is passed through a proxy which monitors labels to ensure that the security levels of both endpoints of the IPC are compatible with the labels of the data being transferred between the two processes. This is discussed in more detail in the related works chapter. In Data Tethers, we would take a slightly different approach, since we are not interested in security labels in the classic sense of controlling flow from secure to insecure contexts, but instead with ensuring that labels are properly propagated when data is transferred.

For pipes, this is fairly straightforward to implement. When labeled data is written to a pipe, we initiate a process similar to policy-controlled file opens and prepare the reading process for data flow tracking. We then keep track in the kernel of which bytes of the pipe buffer are labeled and copy the labels into the receiving processes shadow space on reads. Since labels bits between processes are not necessarily the same, we must translate the bit vector to match the label-

to-bit-map in the reading process.

Semaphores do not transmit data directly, but could be used to transmit bits between processes via a timing channel. While we are not concerned with covert channels in Data Tethers, it might be possible to prevent this. Since (POSIX) semaphore functions have no parameters, data leakage would occur via implicit flows, i.e. conditionally calling the function based on one or more labeled variables. By synchronizing the implicit context of both processes' involved so that they share the combination of both processes implicit context when calls to sem_post() and sem_wait() are made, we can transfer the proper labels for this implicit flow across processes.

Finally we consider the case of shared memory pages. Shared memory pages are a single physical page mapped into both processes address space. Because either process can modify the page simultaneous, it is difficult to maintain label integrity between the processes. The simplest solution would be to unmap the page in both process, so that access to the shared page would generate a page fault. In the page fault handler, we could then enforce a policy of allowing only one process at a time to have the page mapped into memory, preventing simultaneous access. We could also update each process's shadow page for the shared page, when mapping the page into the requestor's address space.

6.2 System Calls

There are several places in the Data Tethers system where user space processes need to communicate with the kernel. To provide this functionality, we added system calls to the kernel. These system calls included:

- sys_enableWatchpoints - turn on watchpoints for labeled data regions. Called at the end of an instrumented function.

- sys_disableWatchpoints - turn off watchpoints for labeled data regions. Called at the beginning of an instrumented function.

- sys_trackingReady - notifies the operating system that the instrumentor has initialized and the target process can be resumed.

- sys_policyStatus - used by the policy monitor to notify the operating system of the status of a policy. This could be either the result of a verification request from the operating system, or a policy violation.

These system calls first check to make sure that the calling process is correct. For example, only the policy monitor is allowed to call the sys_policyStatus system call. When other processes call these system calls, the operating system simply returns.

6.3 Interfacing with the Policy Monitor

In order to request policy verification, receive decryption keys, and to be notified of policy violations, the operating system must interface with the policy monitor (discussed in detail in Chapter 8. In this section, we will discuss the mechanisms it uses to do that, as well as the steps involved in policy verification and policy violation.

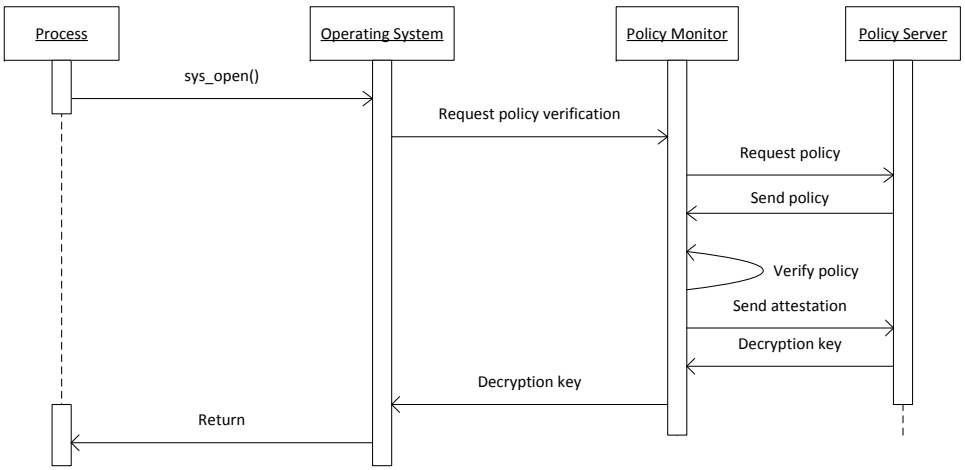

Figure 6.3: Operating system requesting policy verification

6.3.1 Policy Verification

When a process opens a policy-controlled file, the operating system must verify that the policy conditions are met, and request the proper decryption key. The system component responsible for this is a user space process called the policy monitor. The process by which this happens can be seen in figure 6.3.1.

In order to request verification for a policy, the operating system creates a file containing the policy ID in a folder which is monitored by the policy monitor. If the policy monitor does not currently have the policy requested, it requests it from the policy server, then downloads the modules it needs to verify the policy. After verification, it sends an attestation to the policy server, receives the encryption key, then notifies the operating system of the status of the policy using the sys_policyStatus() system call.

6.3.2 Policy Violation

When a policy is violated, the policy monitor notifies the kernel via the sys_policyStatus() system call. The kernel takes immediate action. First, the kernel suspends all processes affected by the policy change, then encrypts their affected memory. After it deletes the encryption keys for the violated policy from memory, it flushes any devices, such as the video or sound card, that have had labeled data written to them. We encrypt swapped pages containing labeled data with the appropriate keys, so once the keys are destroyed these pages are secure. Processes are currently killed if they have affected data, since continuing with encrypted data may make them unstable. Future implementations will allow the user to decide if he wants to continue running the application or suspend it to disk.

6.4 Data Tethers Processes

In order to keep track of policies and labeled data on a per process basis, some changes were required to the OpenSolaris process data structures, and some additional code was added to manage watchpoints, policies, and label bits. This section will discuss these changes.

6.4.1 Watchpoint Management

In order to trigger the instrumentation of functions, Data Tethers relies on watchpoints on labeled data to detect when this data is accessed. Managing these watchpoints in an efficient manner is key to the performance of the Data Tethers system.

Watchpoints can be handled either in hardware or software. While hardware is much more efficient, typically processors are limited to one or two watchpoint addresses. Because of this, watching more than a couple of addresses requires software watchpoints. Software watchpoints are implemented by setting page permissions to non-readable and non-writable, and then looking up the violating address in a table. If the address is not being watched, the page permissions are set to readable and writable, and the process is single stepped by the kernel. The page permissions are then restored to non-readable and non-writable. While some operating systems rely on user space implementations of a watchpoint system, OpenSolaris includes software watchpoints as part of the oeprating system.

One problem Data Tethers must address is how to maintain these watchpoint tables in an efficient manner, adding new watched regions as needed, and removing regions that no longer need to be watched. We also need an efficient way to disable and enable watchpoints as we enter and leave safe (instrumented) functions.

When a policy-controlled section of a file is read in by a process, the kernel sets watchpoints on the portions of the read buffer than contains that data. When a watchpoint is hit, the kernel sends a SIGBRK to the process, which then triggers the instrumentation of the currently executing function. This function will then continue executing with watchpoints disabled, copying labels along with the data.

At the end of the function's execution, it calls the sys_enableWatchpoints system call. At this point, the operating system must set new watchpoints on newly labeled regions. In order to do this efficiently, we traverse the page table structure of the shadow memory segment and look for pages that are marked as accessed, then scan each modified page looking for regions where labels have

changed from zero to non-zero or from non-zero to zero by comparing against the currently watched regions. This scanning is costly, but because we only look at pages that have been modified in this function, the number of pages we must scan is minimal. Because functions may be interrupted before completion and page access bits reset by the scheduler, we keep set a flag that tells us if watchpoints are currently enabled or disabled; the scheduler maintains a list of dirty pages if they are disabled.

Of particular concern, for efficiency regions, is the call stack. Generally speaking, labels are only removed when constant values are assigned to previously labeled memory addresses, for example when zeroing out a region of memory. However, for the call stack when a function exits it may leave labeled data on the call stack without zeroing it out, even though it is not used. To correct this, we insert code before making the system call to zero out the both the label of stack variables and the stack variables themselves in the existing stack frame. If we did not do this, stack variables would generate large numbers of spurious page faults due to labels on data that is part of function calls that have exited.

6.4.2 Policy Management Structures

In order to keep track of policies in use in the system, we added several data structures. There are a couple of different use cases that we needed to cover, so we keep some slightly redundant data, for the purpose of efficiency.

At the process level, we need to keep track of the policies currently in use, and the bit in the label bit vector that is assigned to those policies. This is a relatively small number of policies, so we keep a simple linked list that contains the policy

bit, and a pointer to a shared struct that contains the policy information. The goal of this list is to provide quick lookups from policy ID to bit or bit to policy ID when performing reads or writes of labeled data.

At a global level, we need a data structure that can allow us to quickly look up all processes that use a particular policy, in the event that policy is violated. This is crucial to quick cleanup on violation and reducing the window of attack. For this, we keep a global table, hashed by policy ID that contains a list of pointers to the process structs of all processes that use that policy. In the event of a violation, the kernel can quickly locate these processes and clean them up.

6.4.3 Label Space Segment

In order to store labels, we needed to allocate a special segment to contain the shadow pages. The version of OpenSolaris we are using is a 64 bit operating system, with 48 significant bits of address space. This give us more than sufficient space to allocate this segment. We chose an address range for the segment that is between the segment used for heap and the "hole" of unusable addresses, as can be seen in Figure 3.1. When a process opens its first tethered file, this new segment is added to its address space for shadow pages.

CHAPTER 7

Policy Language

In order to specify the conditions under which data in the Data Tethers system can be accessed, it is necessary to have a policy language that is sufficiently powerful and flexible to express these policies, but which is simple enough that new policies can be easily written without introducing bugs or errors due to its complexity. In this chapter, we discuss available policy languages and their various strengths and weaknesses, and then specify the Data Tethers policy language.

While the choice of a proper policy language might seem trivial, it is in fact very important to the security of the Data Tethers system. With the Data Tethers system, we aim to allow arbitrary policies and modules to evaluate those policies. If the policy language is too simple, it is impossible for us to specify the acceptable conditions for the module to evaluate. Even complex languages may not be able to adequately specify the types of environmental conditions required of Data Tethers policies. In addition, very complex languages make it difficult to *correctly* specify the policy without making errors, and are generally not human readable. While tools can be created to make creation of these policies simpler, the complexity of the language leaves room for flaws in policies due to bugs in the software.

7.1 Existing Policy Languages

The eXtensible Access Control Markup Language(XACML) is the most widely used access control language. XACML is an XML based Attribute Based Control System, along with a standardized processing model for processing the policies[20].

The highest level organizing construct for XACML is the <PolicySet>. <PolicySet> elements consist of other <PolicySet> elements, or <Policy> elements along with a method for combining the evaluation results. <Policy> elements consist of a set of <Rule> elements, and a method for combining them. <Rule> elements are Boolean expressions that are combined to form an **authorization decision.** All three of these elements can contain <Target> elements, which include a 3-tuple of requestor, resource, and action.

Combination methods include:

- Deny-overrides - denies if any single element evaluates to false
- Permit-overrides - permits if any single element evaluates to true
- First-applicable - evaluates to the first element whose target and condition is applicable to the decision request
- Only-one-applicable - One and only one policy should be applicable, based on the target

XACML also include the concept of an *obligation.* An obligation is a directive or requirement that must be carried out prior to granting access. This is different from a formal requirement for access in that it is an action that must be taken,

rather than a condition for access. An example of this might be that the access must be logged to the system log when access is granted.

In addition to a specification, XACML includes a standard evaluation model, along with several components. These components include [32]:

- Policy Administration Point(PAP) - manages policy
- Policy Enforcement Point(PEP) - enforces policy
- Policy Decision Point(PDP) - evaluates policy and issues authorizations
- Policy Information Point(PIP) - external information source for the PDP
- Policy Retrieval Point(PRP) - stores policy

While this brief description of XACML may sound completely manageable, the four elements discussed above are only four of 58 total elements available in the specification. The full specification is 154 pages long and includes many, many options and generalizations designed to handle a wide variety of applications. While we felt that it was possible to implement the Data Tethers policies in XACML, the complexity of the specification was excessive for the specific types of policies we wanted to create, and that this complexity would make it difficult to make an easily usable and administratable system.

7.2 Data Tethers Policy Language

Data Tethers policies are written in an XML-based language that is significantly simpler than the XACML language. The schema for our policy language is as follows:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="Policy">
  <xs:complexType>
    <xs:attribute name="policyid" type="xs:positiveInteger"/>
    <xs:attribute name="cacheable" type="xs:boolean"/>
    <xs:attribute name="CanFreeze" type="xs:boolean"/>
    <xs:attribute name="PollIntervalSeconds" type="xs:positiveInteger"/>
    <xs:all>
      <xs:element name="Access" type="AndOrType"/>
      <xs:element name="Actions">
        <xs:complexType>
        <xs:all>
          <xs:element name="ActionID" type="xs:string"/>
          <xs:element name="Conditions" type="AndOrType"/>
        </xs:all>
        </xs:complexType>

      </xs:element>
    </xs:all>
  </xs:complexType>
</xs:element>

<xs:complexType name="AndOrType">
<xs:element name="AND" type="AndOrType" minOccurs="0"/>
```

```
<xs:element name="OR" type="AndOrType" minOccurs="0"/>
<xs:element name="Module" type="ModuleType" minOccurs="0"/>
</xs:complexType>

<xs:complexType name="ModuleType">
  <xs:attribute name="ModuleID" type="xs:string">
  <xs:attribute name="Expect" type="xs:boolean">
  <xs:element name = "Parameters" minOccurs="0">
    <xs:complexType>
      <xs:element name="ParameterName" type="xs:string"/>
      <xs:element name="ParameterValue" type=:xs:string/>
    </xs:complexType>
  </xs:element>
</xs:complexType>
</xs:schema>
```

As you can see, our schema is significantly simpler than XACML. Our goal in the design of this language was to construct a language that let us specify arbitrary boolean logic expressions, with the variables in the expressions being output from modules. Because we allow arbitrary modules to be implemented, we needed a flexible parameter passing mechanism. Towards this goal, we designed a structure composed of AND-OR nodes, with each module having an "Expect" parameter that allows us to specify if this module should return a true or false, effectively giving us a NOT operation. Modules can take an arbitrary number of name-value pairs as inputs, which are also specified in the policy document.

Each policy in our language begins with the policy ID, followed by the tags control usability features of the policy. These tags include:

- Cacheable - can the encryption key be cached for when the computer does not have network access? This is, of course, very insecure, but could be necessary if the device must operate in an environment without network access
- CanFreeze - can we freeze applications using data with invalid policies and persist them to disk? This option allows the operating system to encrypt the in-memory image of the application before destroying the key. This takes significantly longer than just clearing memory and destroying the key, but allows for increased usability
- Polling interval - how often should we verify this policy is valid?

The next section controls general access to the data, followed by a series of controlled actions for the data, such as read, copy, print, etc. Each of these sections contains a list of modules and required parameters for those modules to allow access or the particular action. Modules represent testable conditions, such as location, time of day, or user identity and may be combined in AND/OR relationships. For example, we may allow printing if we are in the office location and the printer we are using is in a set of secure printers.

Figure 7.1 shows a simple Data Tethers policy for a fictitious ABC corporation. It has requirements for accessing the data, as well as printing it. The data can be accessed if the computer is not connected to public WIFI, and the owner of the laptop has his RFID tag in range of the tag reader, presumably to require

```
<Policy PolicyId="1"
       Cacheable="false"
       CanFreeze="no"
       PollInterval="5s" >
  <Access>
    <Or>
      <And>
        <Module ModuleId="PublicWifiTest" Expect="False"/>
        <Module ModuleId="RFIDPresentTest"/>
          <Parameters company=”ABC corp”
                  employeeGroup = “DataTethers”>
      </And>
      <And>
        <Module ModuleId="ABCCorpLocationTest" />
      </And>
    </Or>
  </Access>
  <Actions>
    <Action Id="print">
      <Module ModuleId="ABCCorpLocationTest"/>
    </Action>
  </Actions>
</Policy>
```

Figure 7.1: Example of a Data Tethers policy.

that he be physically at the computer. The RFID module also has parameters specifying that the RFID should belong to the appropriate company and that the employee is from the proper group. The data can also be accessed if the computer is located in the company offices, with no further requirements. Printing is restricted to when the computer is physically located on company premises. The polling interval specifies how often the policy should be checked and represents a trade-off between the cost of testing the state of the computer and the length of an attack window, should a policy be violated.

To insure that policies are legitimate, all policies are signed by the Policy Server(to be discussed in the following chapter.)

CHAPTER 8

Policy Management Subsystem

In this chapter we will discuss three pieces of the Data Tethers system that deal with policies and work closely together: the policy monitor, policy server, and the policy database, and some additional components that support these pieces. In the terminology of XACML, these would be equivalent to the Policy Administration Point(PAP), the Policy Enforcement Point(PEP), the Policy Information Point(PIP), and the Policy Retrieval Point(PRP), though the mapping between the Data Tethers and XACML models is not one-to-one.

Before we discuss the details of the implementations of these components, we will first give perspective by discussing how they work together to manage and enforce policy. In Figure 8.1, we can see the major components of the policy management subsystem and how they are connected. The policy monitor resides on the end user machine, and uses the policy evaluation modules as PIPs to determine if the policy is valid. The policy evaluation modules are designed to test one individual rule in a policy. The policy monitor is connected to the policy server, which provides it with policy documents on request, accepts signed attestations that the policy is valid, and returns encryption keys once the policy monitor has proven the policy is satisfied. The policy server is connected to the policy database and module repository, which are the central storage points for

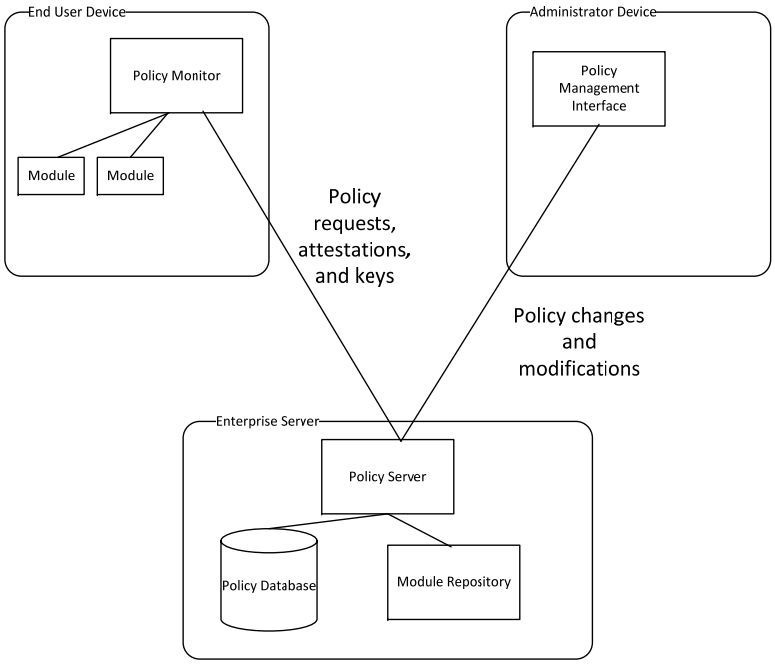

Figure 8.1: Policy Management subsystem.

policy documents and policy evaluation modules, respectively.

We also see in this diagram that the policy administration tool connects to the policy server and allows the data administrator to create new policies on demand. This tool is also the primary way to attach policies to new data. While this tool is not a core part of the system, it is quite useful from a practical perspective.

8.1 Policy Monitor

The policy monitor is the part of the Data Tethers system that is directly responsible for environmental monitoring. It runs at higher than root privileges on the Data Tethers machine, but within a restricted environment. When a data controlled by a new policy is first encountered by the operating system, it sends

a verification request to the policy monitor for the policy ID. The policy monitor then requests the policy from the remote policy server, downloads any modules necessary to verify the policy, verifies that the current system state meets the policy's requirements and sends a signed attestation of this to the remote policy server. This signed attestation includes certifications for the entire software chain, including bootloader, operating system, and policy monitor, as well as the signed results of each module, to ensure that the attestation cannot be falsified (except in the case of a security flaw in the software.)

The policy server verifies the attestation and forwards the decryption key to the policy monitor, which hands the key to the kernel for data decryption. The policy monitor then begins monitoring the system state at the intervals prescribed in the policy. If the state becomes inconsistent with the policy, the monitor notifies the kernel, via a system call, which begins the cleanup process.

As mentioned before, the policy monitor is designed to allow for arbitrary modules to be executed. When a new module is encountered, it is downloaded from the policy server. A common interface is published that modules must implement, allowing them to be loaded at run time. Each module is designed to test one particular rule in a policy. For example, location or time of day.

8.2 Policy Server

The current policy server is a relatively simple application connected to the policy database and module repository. It has two interfaces: one for the policy monitor and one for policy administration, both using TLS for security. The policy monitor interface has three main functions:

- Policy requests
- Attestation processing
- Encryption key transmission

Additionally, the policy server can notify the policy monitor when policies or modules are updated in the policy database, allowing changes to quickly be sent to client machines.

The second interface the policy server provides is for policy administration. This interface provides lists of policies, and the ability to create new policies, either from scratch or by combining existing policies. Additionally, the administrative interface allows for the upload of new or updated policy modules. As an aside, an administrative tool that uses this interface was created as part of a Master's degree project.

8.3 Policy Database and Module Repository

The policy database is an SQL-based database, the schema for which can be seen in 8.2. The database contains several normalized tables to allow for efficient combination of policies without duplication, and to allow for updates to policy components to affect all relevant policies. The PolicyElement table is where individual elements of a policy are defined. The PolicyElementType can be "Policy", "AND", "OR", or "Module".

A many-to-many relationship between policy elements is possible, and is stored in the PolicyElementRelationship table. This table allows a policy to be hierarchically defined, and also to share components with other policies. The

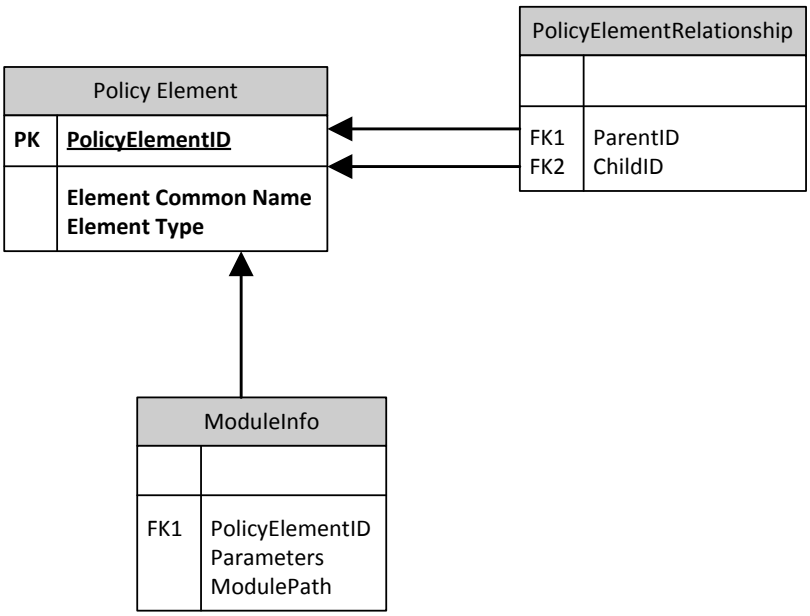

Figure 8.2: Policy database schema.

parent-child relationship imposes a tree structure on the overall policy. Children can be of any type, including other policies, with leaf nodes being modules.

The final table is the ModuleInfo table. It includes a list of all modules available, including their parameters, and links to their locations in the file system. PolicyElement rows with type "Module" will have a corresponding entry in this table. The module repository is a simple file system-based repository. Modules are kept as shared libraries with the version names attached, and are delivered to the policy monitor on demand.

8.4 Environmental Policy Modules

8.5 TPM Limitations

Our current implementation is limited in that we do not have access to a TPM module for the Sun server we are using for development, so we cannot verify that the bootloader and kernel are trusted. Because of this, in many locations in this dissertation we discuss attestations where the current implementation does not have complete code for this functionality. While we do transmit dummy attestations, they are not checked by the policy server or any other component. However, trusted software chains are well known, and their implementation based on a hardware TPM is well known and simple to do.

CHAPTER 9

Performance

The most crucial evaluation issue for Data Tethers is that it functions properly and that labels are correctly propagated. The cost of running instrumented applications is also important, particularly from a usability perspective. We break this down into several components: cost of rewriting code, cost of running instrumented code, cost of file system changes, cost of extra memory pages, and cost of watch points. We also must evaluate the speed of cleaning violated policy data.

9.1 Test Setup

Our test setup for these experiments was a Sun T2000 server with four UltraSPARC T1 processors and 32 gigabytes of memory. Each operating system instance was run in a separate virtual machine using the Sun Logical Domains(LDOM) and the Sun Hypervisor. Each virtual machine was allocated 8 cores (one physical processor) and 8 gigabytes of memory, along with a dedicated partition on the hard drive. The hard drive used was a 500 gigabyte, 10k RPM SAS drive which was used *only* by the virtual machine currently being tested.

The baseline operating system used in the tests was an unmodified installation

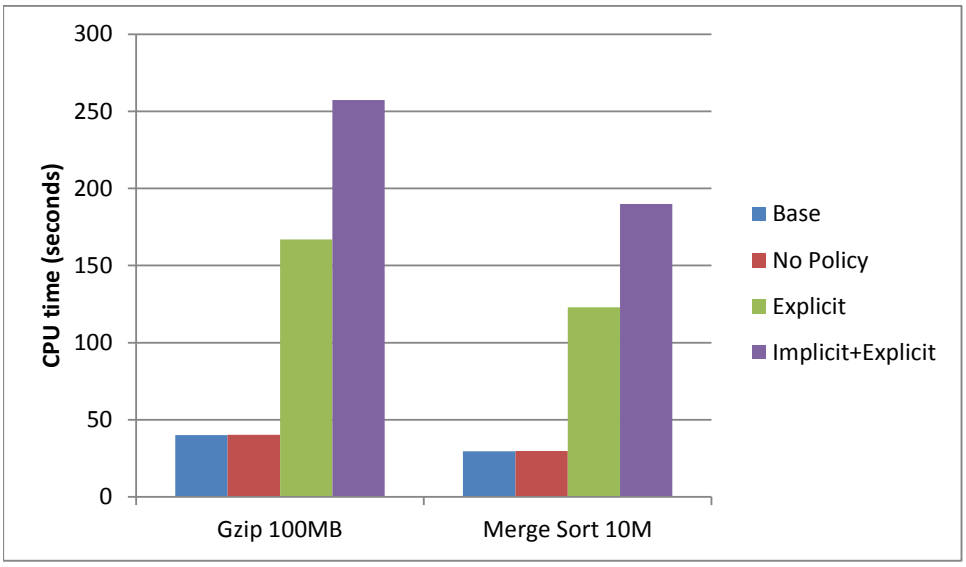

Figure 9.1: Micro benchmarks.

of the base OpenSolaris system used for the Data Tethers system. A custom kernel was compiled, using the same compilation options as the Data Tethers kernel to avoid any unforseen optimization or undocumented build switches in the Sun compiled kernel.

9.2 Costs of Running Instrumented Code

9.2.1 Micro Benchmarks

To evaluate the cost of running rewritten code, we tested both microbenchmarks and user application benchmarks. Our microbenchmarks are iterative versions of merge sort and gzip. These applications work almost exclusively with data and are instrumented in their entirety, representing the worst-case scenario for Data Tethers. We tested gzip with multiple file sizes, from a few kilobytes to

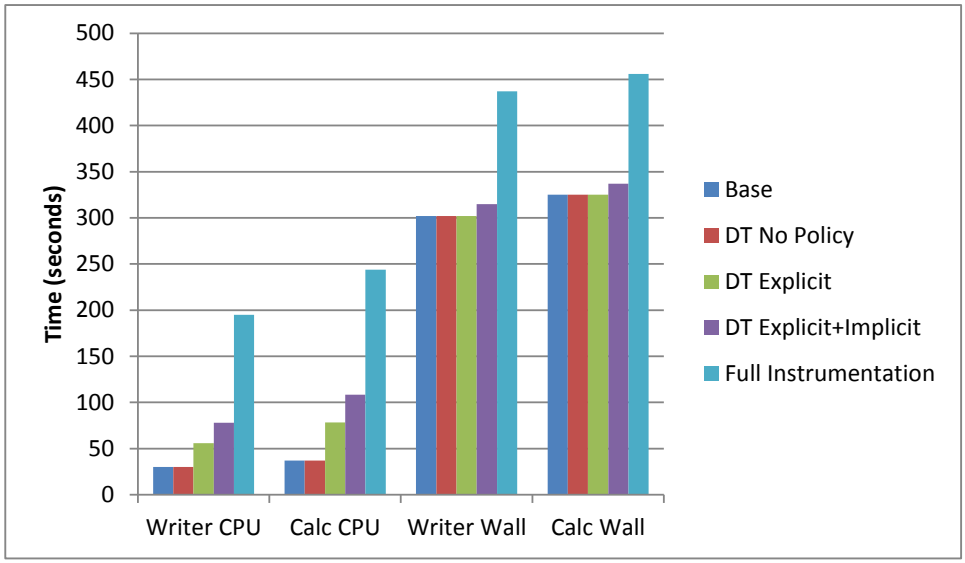

Figure 9.2: Application benchmarks.

several megabytes and merge sort with labeled input files from 1000 elements to 1 million elements, and averaged run times over multiple runs. These results are shown in Figure 9.1. For explicit only tracking, the increase in running time was 4.1 times the base CPU time. For explicit and implicit combined, this increased to 6.4 times the base CPU time. This increase was quite large, but was expected, considering the number of instructions added in the instrumentation process, increased cache misses, and other side effects.

9.2.2 User Applications

For application tests we chose two representative user applications: OpenOffice Writer and Calc, the OpenOffice word processor and spreadsheet. Typical use of these types of applications is hard to define, but we attempted to capture representative real user interactions using record and playback tools. In order to

record our test cases, we installed the Cygwin X server on a Windows 7 computer. We then recorded our tests in a piece-wise fashion, for example clicking on the file window, then waiting for the tool to recognize that the menu had opened before clicking on the open button, waiting for keystrokes to appear before type the next one, etc. This is to ensure that we accurately count the time required to complete the task, rather than a simple fixed duration script.

For Writer, we recorded opening a document, adding several pages of text, removing text, cutting and pasting from one document to another using the internal copy and paste buffer, and saving the file. For Calc we entered numbers in a spreadsheet, copied and pasted numbers from one sheet to another, triggered formula updates across linked sheets, did simple calculations such as averages over ranges of numbers, and saved the spreadsheet. We played this recording back, measuring CPU time and wall clock time, averaged over multiple runs, with the results seen in Figure 9.1. All data in document used in the benchmarks was tethered, though fonts, configuration, and other files were not.

Though CPU time use increased by 2-2.5x, wall clock time is only increases by less than 5%. The extra clock cycles used were mostly idle, and given that the machine used for these tests is a Sun T2000 server with an ULTRASparc T1 processor, with each core comparable to a Pentium 3 1 Ghz desktop processor, we feel that the performance would be acceptable on any modern processor.

9.3 Costs of Rewriting Code

One potential performance bottleneck is the overhead imposed by Dyninst for parsing and rewriting the executable code. In our experiments with application

benchmarking, we found that the the cost of rewriting code is low. The largest cost is Dyninst's initial parsing of the in-memory image of the application and shared libraries, which depends on the size of the application.

The largest application tested, OpenOffice Writer, was around 100 megabytes in executable code size, including all libraries, and took less than 3 seconds total CPU time to instrument. Once instrumented, binaries could be saved and reused to limit this overhead.

9.4 File System Benchmarks

For the file system, we performed a series of microbenchmarks to quantify the costs of our modifications to tethered and untethered files. Figure 9.3 shows the results of our tests for both reading and writing both types of files. Care was taken to eliminate caching effects by using different files for each iteration of the test.

Data Tethers slows down the file system, primarily due to the need to scan memory for tags and to encrypt/decrypt. Approximately the same overhead is added to both reads and writes, but the base time for writes is much lower than for reads. Reads must wait for the disk read to complete before returning to the user, while writes may be completed asynchronously. Numbers for larger files are linearly related by size to the number shown for 16k files. Overhead is added for opens because an additional check is done on open to determine if the file is tethered. For most user applications, the overhead introduced is not noticeable compared to the run time of the application, but applications that open and close many small files may see a performance decrease.

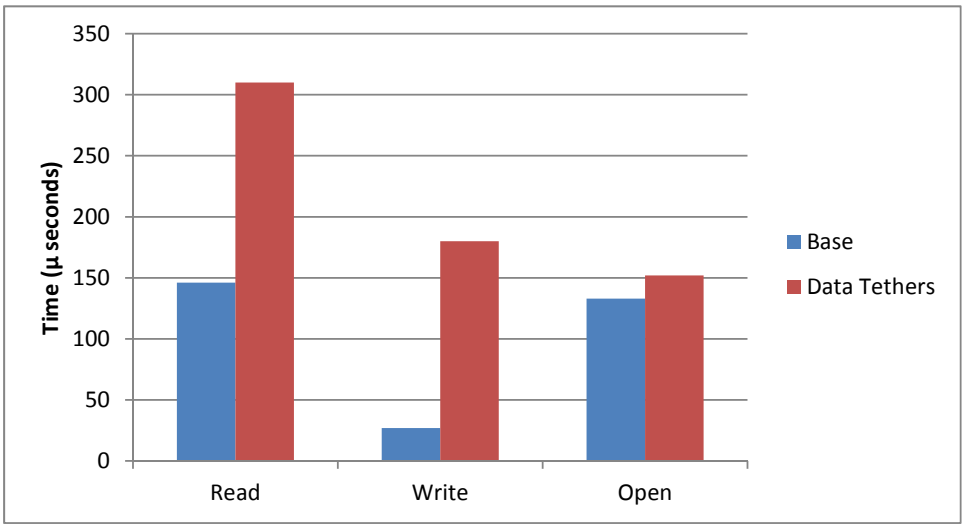

Figure 9.3: File system benchmarks.

9.5 Network Benchmarks

For the network, we are mainly interested in the overhead of packing and unpacking tethered data, so our benchmarks measured time spent in the appropriate SockFS functions, not transmission time. Since network traffic always requires scanning for start tags, we benchmarked the Data Tethers system both for data with and without a policy attached. For these benchmarks we computed average time spent in the SockFS send and recv functions for small buffers of 256 bytes.

Figure 9.4 shows that the cost of sending a packet of tethered data is approximately twice that for a nontethered packet, primarily due to scanning and encrypting the data. In Figure 9.5, we see receiving tethered packets costs slightly more than three times as much as an untethered packet. This difference between send and recv is due to buffering partially received decryption blocks. Very network-intensive applications may be negatively impacted by Data Tethers, but

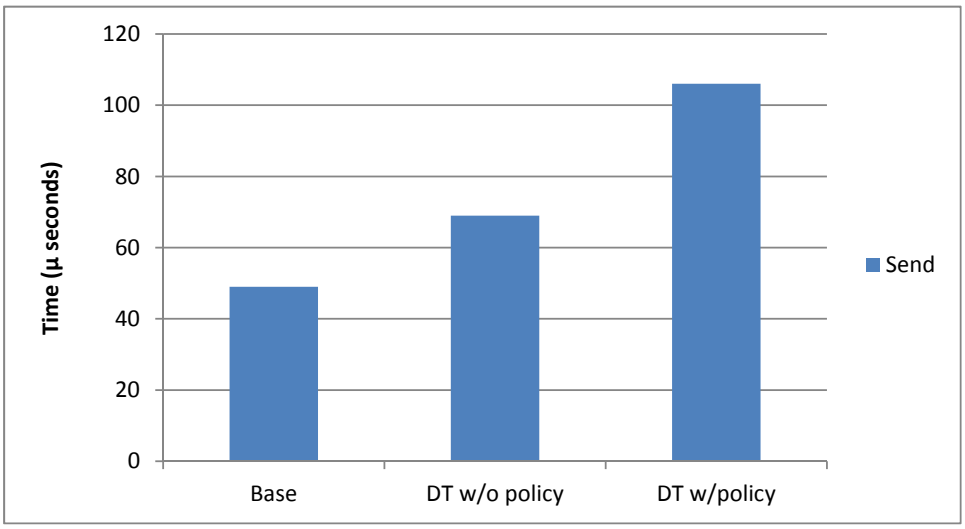

Figure 9.4: Network send benchmark.

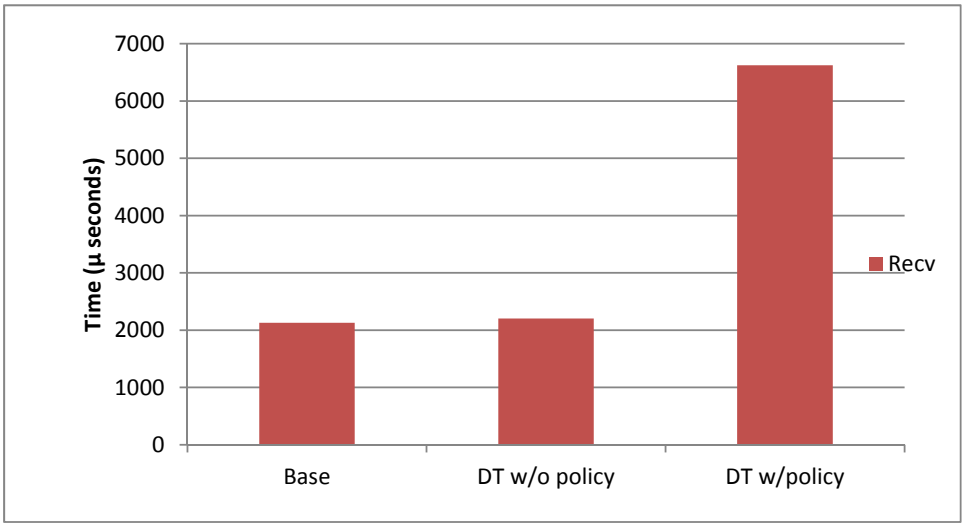

Figure 9.5: Network recv benchmark.

for most end-user applications, the overhead introduced is dwarfed by transmission times.

Also of interest is the data size overhead introduced when tethering streams. This varies depending on the degree to which the data is tethered. In the best case, where a stream is tethered with a single policy, the overhead is around 100-200 bytes. In the worst case, where every word is tethered with a different policy, or even multiple policies, this could balloon to 100-200 bytes per word. We anticipate that the worst case is very unlikely, and that most streams will contain few or no policies. If this is not the case, it would drastically improve network performance to increase the size of the send buffer, and send data similar to the file system data structure in a side-channel.

9.6 Paging and Page Protection Faults

Calculating the number of extra pages used by tethered applications is straightforward since one page is allocated for every page of tethered data. For applications like merge sort and gzip, there is one extra page per page of data, since all data is tethered. For the OpenOffice applications, it depends on the amount of internal data structures that are generated by opening the document. We found on average that for a 1 megabyte document where all data was tethered, around 3 megabytes of shadow memory physical pages were generated.

Also of interest is the number of page protection faults generated by the application. Data Tethers relies on these to trigger instrumentation, but each page protection fault triggers a switch to kernel mode, and excessive page faults has a serious performance impact. For gzip and merge sort, after the initial round

of watch point faults, no additional page protection faults were generated since all code was instrumented at this point. For the OpenOffice applications, a small number of page protection faults were generated due to the application accessing non-watch pointed data on pages containing watch points, but the number was quite small, 0.4 faults per second on average. Presumably these applications use slab allocators to reduce memory fragmentation, minimizing the amount of mixed data types on physical pages.

9.7 Speed of Cleanup

To measure speed of cleanup we opened several applications with labeled data, and then forced a policy violation. Timing code was added to the kernel to measure the amount of time from the initial notification of the violation until the completion of cleanup. This took, on average, less than 100 milliseconds.

9.8 Correctness

In terms of correctness of the implementation, we are primarily interested in whether or not labels are correct in most scenarios. Our explicit flow implementation is similar to other systems (discussed in Chapter 10). While we did not do a formal proof of correctness, we strongly believe it is correct, barring implementation bugs. A formal proof of correctness would be quite involved given the size of the SPARC instruction set. As mentioned in Chapter 4, it is impossible for our implicit flow tracking code to be 100% correct in all instances, but we are specifically interesting in where it results in incorrect labels in practical cases.

We tested multiple cases, ranging from copying data from place to place in memory to combining data with multiple policies in various ways. All of the performance benchmarks discussed in the following sections were also checked for correctness. In all cases file labels were correct, even after multiple iterations of the tests.

We also verified that data with no policy attached was written unencrypted when the writing process had policy-controlled files open, and that data and keys were correctly cleaned up when policies were violated. It is our feeling that for almost all practical cases, our implementation will result in correctly labeled data.

CHAPTER 10

Related Work

There are two main areas of research in Data Tethers: environmental policy based access control, and dynamic information flow tracking. Both of these areas have been studied extensively. In this chapter we will discuss related work in these two areas. Because of the volume of research, it may not be possible to cover every related paper, but we will attempt to cover the most important work and major themes in each area.

10.1 Secure Systems

The history of secure operating systems is long and varied. In this section we focus on operating systems that implement policy based access control mechanisms, as well as operating systems that integrate information flow control.

10.1.1 Policy Based Access Control

The FLASK operating system is a micro-kernel based operating system architecture, implemented in the Fluke operating system, that focuses on separating access control enforcement from the access control policies themselves. Enforcement is done based on labels, which are handed out by a security server based on

policy decisions. Access control is at the object level, where objects are typically operating system constructs such as pipes or files. One key FLASK design feature is the ability to revoke access to any object when policies or conditions change. While FLASK and Data Tethers share some common features, particularly with respect to the goals of flexible policy enforcement, FLASK is focused on access control on IPC and to external resources such as files or the network, whereas Data Tethers is focused on access control to data itself. For example, FLASK can revoke access to a file that it previously allowed reading from, but it cannot revoke access to the data that came from that file. It also provides no information flow tracking, remote policy server, or encryption to protect data in the event the machine is lost.

SELinux is an implementation of the FLASK architecture using the Linux kernel. SELinux is was originally implemented as a modified kernel, but now is a series of kernel modules and user space tools for managing policies and process permissions [62]. Like FLASK, SELinux is focused on separating policy decision from enforcement, and on providing flexible policies. A similar competing system is called AppArmor[61].

[26] details a posture-based security system which does environmental monitoring similar to Data Tethers, but this system only limits what applications may be run under various conditions.

10.1.2 Information Flow Control

The Asbestos operating system implements a decentralized, fine-grained, mandatory access control system which allows processes to label data and create com-

partments of the main process to handle data with that label. It can then apply access control policies to these labels, which are enforced by the kernel[27]. This prevents data loss due to external attackers or mis-configured software by disallowing flows through channels that are not authorized. Data Tethers differs from Asbestos in that it focuses on protecting mobile data, not preventing data loss through excessive process privilege.

Histar [72] builds a Unix-like environment on top of an information flow control system using labels similar to Asbestos. This allows the construction of applications with explicit information flows between components. The DStar system [73] extends the Histar model to distributed systems, allowing for transmission of labeled data between machines, however it is limited to tightly coupled machines in a cluster environment. Again, while the goal of Histar is to prevent data loss, it is focused on protecting server applications from malicious external attacks, and it provides no special protection in the case of physical loss of the device or against the types of physical attacks Data Tethers is intended to defend against.

Additionally, Histar requires applications to be rewritten from scratch, whereas Data Tethers can work on existing binaries. Histar also uses much coarser tainting methods than Data Tethers. Histar taints at the file level, and combinations of files receive the highest taint level of any file involved. In Data Tethers, a combination of files would have be labeled at the byte level, and could be split apart at a later time into pieces with different policies. A good example of when this might be useful is with zip files.

There have been two recent virtual machine based approaches that are similar in nature to DT, the S3 system[39] and the Neon system[74]. Both of these sys-

tems run the operating system inside of QEMU and convert native x86 assembly instructions into simulated instructions that do information flow tracking. While these are similar in flavor to DT, they suffer from several drawbacks in comparison. Since the information flow tracking happens at a higher level than the operating system, it is difficult to detect the types of environmental conditions we use in Data Tethers, for example, that a virus scanner is running or a specific user logged in. Also, since the operating system itself is instrumented it is possible for unrelated processes to become tainted, particularly since the operating system, by its very nature, is a long running process.

Trishul is a Java virtual machine based approach that implements both explicit and a conservative implicit flow tracking[47]. Similar to Data Tethers, Trishul can enforce usage policies on this data. However, Trishul was not designed for mobile data protection, has no mechanisms for protecting data when policies are violated, and from all indications was never completed. In addition, the implicit flow tracking method Trishul uses attempts to determine all variables that could be modified inside of a control block, and when it cannot, falls back to globally labeling all following assignments with the condition variable label. In practice, this happens quite often, limiting the usefulness of the system.

PinUP is a similar access control overlay system to Data Tethers. PinUP extends the standard Unix file access model by allowing the administrator to specify specific applications that are allowed to access a particular file. The goal of PinUP is to define and protect application workflows by limiting access only by approved applications.

A data provenance system called GARM [24] uses static analysis to track provenance information. CLAMP [49] is a system with similar goals of preventing

data leakage, but focuses on web applications using a LAMP-style development stack.

10.2 Information Flow Tracking

Information flow tracking has been studied both theoretically and practically since the mid 1960s; one classic theory paper is Denning and Denning [25]. Recently, there has been new interest in practical information tracking implementations in both hardware and software. A more recent discussion of the current state of information flow tracking can be found in [56] and a discussion of the limits of taint tracking, and implicit flow tracking in binaries can be found in [16].

A core technique for implicit flow tracking is the idea of linear continuations. This concept was introduced by Zdancewic and Myers in [71], and forms a key component of most implicit flow tracking systems, including Data Tethers. While in the naive case, when a labeled control flow variable was encountered, the program counter label would be increased if the label of the variable was higher than the current label of the PC, and no mechanism would exist to lower it. Linear continuations allow us to identify when two branches of a control flow decision merge into a common point. By creating a stack structure, we can push the label of control flow variables on the stack at a branch(if they are higher than the current PC label) and pop them off when the branches merge. This prevents significant over-labeling in the Data Tethers case, or rejection of programs that satisfy non-interference in the multi-level security case.

10.2.1 Hardware Based

Many hardware solutions have been proposed for information flow tracking. RIFLE [66] uses labels that are stored adjacent in memory to data, and a special processor that copies and combines these labels as memory is used. RIFLE attempts to handle both implicit and explicit flows by conservatively labeling data via static analysis, though it is difficult to judge how conservative this labeling is, since it is not outlined in detail, and the authors do not take into account undecidable loops.

Minos [22] uses one bit labels to tag data as trusted or untrusted. These labels are stored in orthogonal memory rather than in the processes address space. Trust is based on how long the data has been in the application, with new data being flagged as untrusted. The goal of Minos is to detect control data attacks that overwrite function pointers or return addresses.

Suh *et.al.* independently proposed a similar hardware mechanism for tracking information flows, again limiting the labels to one bit. Like Minos they detect instructions that are themselves tainted as coming from external source, or jumps that have tainted target addresses[59].

Raksha [23] is a hardware implementation that supports 4 bit labels, flexible security policies, and user handled security exceptions. User applications can specify the labels for bytes when receiving data from input channels. Raksha does not attempt implicit flow tracking.

Another proposed system is Flexitaint [67], which stores 2 bit tags in a special memory region with a constant offset from the memory address that is labeled. Flexitaint features many processor level optimizations for handling label data effi-

ciently, flexible taint combination rules, and focuses mainly on providing low level infrastructure for other applications that require DIFT. Like Raksha, Flexitaint does not address implicit flow tracking.

In GLIFT, [64] the authors present a new design for both implicit and explicit flow tracking based on considering information flow at the gate level. While this is an impressive feat, it requires significantly limiting loop behavior to prevent the types of issues mentioned in Chapter 4. In fact, loops are constrained to a fixed number of iterations only, with all unused iteration being predicated on the loop control condition. Because loops all have fixed duration, it is known at run time exactly which memory addresses could have been modified.

Of these various hardware approaches, only Raksha has actually been implemented in hardware using and FPGA. The rest were implemented with QEMU.

10.2.2 Software Based

Many software information flow tracking systems have also been written, mostly focused on taint tracking to prevent exploits.

One interesting approach is that used in TaintDroid[28]. This system tracks the flow of sensitive user information inside Android applications by modifying the Dalvik JVM. While very low overhead was achieved with this approach, it is only applicable to applications that run inside a virtual machine, and not to native binary applications. It also offers no protection in the event a device is lost.

TaintCheck [48] is a system that labels data from untrusted sources, such as network sockets, with a label denoting their source. TaintCheck then instruments

the running application, inserting code to track explicit data flows. It also inserts code at instructions that can be used by exploiters to modify the execution of the application, for example, jump addresses, format strings, or system call arguments, checking to see if they use data from a tainted source.

TaintCheck uses a similar technique for tracking labeling data, but beyond that has few similarities to Data Tethers. TaintCheck protects against external attackers attempting to exploit badly written applications. It offers no protection for mobile data, no type of security policy enforcement, and no tracking of data flows through the system or between applications. TaintCheck also tracks explicit flows only, ignoring implicit flows. It also instruments entire applications, whereas Data Tethers selectively instruments only functions that handle policy-controlled data.

The work done in TaintCheck was later extended to BitBlaze, a series of components for static and dynamic analysis of binary applications. BitBlaze takes a similar approach to S3 or Neon, discussed above, in that it runs the target operating system inside of a specialized version of QEMU. However, BitBlaze is not a secure system itself, but is intended to be a tool that systems that need information flow tracking are built on. While BitBlaze is VM based, it has the ability to parse operating data structures for certain operating systems, giving it the ability to identify individual processes, and flows between them, as well as intercept system calls and similar actions of interest[57].

BitBlaze has very little in common with Data Tethers, aside from the fact that they both use information flow tracking. BitBlaze is a tool, more similar to Dyninst. It does not implement implicit flow tracking, and does not intended to provide any type of security. While it is designed to be flexible in what can

be done with it, it is mostly focused on exploit detection, like most of the other systems in this section.

Built on the BitBlaze platform, DTA++ [38] extends BitBlaze to support implicit flow tracking. However, similar to Data Tethers, they do not attempt to track all implicit flows, noting that malicious programs can thwart any attempts to track implicit flows. Instead, they focus on identifying specific types of implicit flow patterns in benign programs and instrument them in an attempt to reduce the amount of under-tainting due to implicit flows. DTA++ uses a significantly different mechanism than Data Tethers for identification of implicit flows. Instead of relying on run time identification of protected data access, it uses a series of test runs which identify under-tainting in output, then dynamic analysis of the binary application, tracing input data corresponding to the under-tainted output to locate causes of that under-tainting to correct them.

Vigilante [21] is a similar system that instruments application code in an attempt to detect Internet worms. Labels are added for each memory location with a label denoting where the data originated. Only explicit data flows are tracked. When a return instruction attempts to load a dirty address, an exception is generated and an alert is sent out over its monitoring network.

Dytan [19] takes a similar approach to Flexitaint, attempting to provide a framework for general information flow tracking by including the ability to specify how labels are propagated, the size of the labels, and allowing the user to identify sources and check points for the labels. Dytan also instruments x86 binaries, and tracks explicit flows, as well as implicit flows, though they do not consider the case of loops, but only consider branch statements.

In LIFT [52], the authors propose an explicit tracking only implementation that attempts to minimize overhead by checking if the block about to be execute contains operations that potentially copy unsafe data to previously safe locations. LIFT uses one bit tags, and attempts to detect buffer overflow attacks targeting program control flow.

Another approach to software based information flow tracking is the use of special compilers that perform analysis on unannotated source code, and generate instrumented binaries. By using high level semantics present in the source code, they are able to achieve impressive performance numbers, though in all cases they only handle explicit flows.

One such system is the General Information Flow Tracking(GIFT) system. In GIFT, the authors develop a special C compiler that inserts explicit flow label tracking into the resulting binaries. This instrumentation can be extended by writing handler functions that implement tag and policy management. Using this technique, the authors are able to reduce the overhead to as low as 1.85 times the base running time.

In [17], the authors take a similar approach, but additionally optimized the code based on the policy that they are trying to enforce. By careful analysis of the code, they are able to reduce the amount of instrumentation they insert dramatically. The overhead of the resulting instrumentation can averages 0.65%.

While these methods yield impressive performance numbers, they are unfortunately not applicable to Data Tethers. In both cases, these systems are interested in protecting against external attackers and exploits, and thus the policies they are dealing with have to do with particular vulnerable instructions. Because

of this, they are able to trim out much of the instrumentation. Data Tethers, however, is tracking more general flows through the system, so this type of optimization is not possible. Additionally, these systems require recompilation of all applications, while Data Tethers does not.

10.2.3 Security Typed Languages

Implicit flow tracking is an active area of research; however, much of this research is of limited use to us since it involves high level programming language abstractions and specialized language constructs. These systems include special security typed languages or language additions for augmenting code with information flow tracking. Sabelfeld and Myers [54] provide a good survey of this work.

The first type of security typed languages we will discuss is typed assembly languages. These are the most applicable to Data Tethers, but still of limited use, since we deal with raw binaries.

The SIF language is a typed assembly language designed to detect both illegal explicit and implicit flows. SIF is designed with traditional multi-level security systems in mind, so it does not combine policies or track taint dynamically. Instead, developers can specify labels for registers and memory, and loads and stores are checked against these labels to ensure that they are consistent with the lattice of security labels[44]. SIF is an improvement on SIFTAL, by the same authors, that required a run-time stack to guarantee non-interference[13].

To handle implicit flows, SIF adds two new meta-instructions which allow a stack based label system for the program counter to be put in place, similar to the Data Tethers implicit stack. At each branch, if the security label of the branch

condition variable is higher than the program counter label, then the new label is pushed onto the stack. When control flow rejoins the main line of execution, this label is popped from the stack. At all times assignments are not allowed to variables with lower security labels than the program counter, thus ensuring non-interference.

SIF is intended as an intermediate language for proving non-interference and does not execute. Instead, it is compiled to a regular assembly language, and the resulting assembly code can be shown to be secure.

Yu and Islam propose a similar language, called Typed Assembly Language for Confidentiality (TALC)[70]. TALC uses type annotations with a RISC-like language which is much more complete than earlier typed assembly languages.

At a higher level of language, several security typed languages have been implemented. The JiF language is a secure typed version of Java, based on the JFlow lanugage proposed in [46]. JiF is the first practical language to allow static checking of flow annotations. JiF incorporates a decentralized label model, similar to the label model that Data Tethers uses, and can use both static and dynamic labels. JiF includes support for verification of non-interference statically and at run-time.

Two similar, but less fully featured languages also exist: FlowCaml [50] and SPARK/Ada[63].

RESIN is another option for high level languages, other than annotated code. RESIN provides programmers with a language runtime and a programming API to allow them to create policy objects that programmers can attach to data. Assertions can be added to these policy objects, which are then tracked by the

runtime's data flow tracking system. Programmers can then create filter objects that define data flow boundaries where assertions must be verified. RESIN is primarily designed to on a trusted server, and assumes non-malicious code. Similar to other systems that target server exploit detection, RESIN does not provide implicit flow tracking.

10.3 Mobile Data Protection

With the increase in mobile devices, there has been a increased interest in protecting mobile data. In this section we will discuss some of these systems that are particularly relevant to Data Tethers.

Several major vendors support remote wipe-out of devices. The Apple iCloud enables users to remotely wipe-out data on a lost or stolen device[37]. However, this feature only works when the phone has access to the network. Disabling the antenna, or turning on the phone in a protected environment will prevent the data from being wiped. Once the phone is running, it is vulnerable to any number of attacks.

Another method to protect data is to utilize cloud-based services. Storing mobile data in the cloud offers several options to protect data. For example, data stored in Google Drive can be configured to require two-step authentication to access. If the device is stolen, the thief does not have access to the authentication device and cannot access data stored in the cloud[31]. This does not protect data already on the device from being compromised, or in the case that the lost device is already logged in. Additionally, this data cannot be accessed without network connectivity, and limited connectivity may have a serious impact on performance.

MacKenzie *et.al.* present a technique called capture resilient cryptography which can be used to protect data stored in the cloud. In this case, data is stored in the cloud, and access is controlled by signing with a private key stored on the mobile device for which the public key has previously been registered with the cloud server. The technique given allows the owner of the lost device to disable the private key in the event the device is lost. Again, while this offers protection for data in the cloud, it offers no protection for data stored on the device[43].

Another technique that is similar to Data Tethers location based policy module is Mobile user location-specific encryption (MULE)[58]. MULE protects mobile data by using Trusted Location Devices(TLD), installed in locations where the user can legitimately access the data. By communicating with the TLD and utilizing data only available via a constrained channel, such as an infrared beacon, the device can generate keys to decrypt the requested data. While this is an interesting system, it only addresses the location aspect of data protection. There are many scenarios where this is either too restrictive, or not restrictive enough. Data Tethers offers much superior protection.

Keypad[30] is a file system that tracks file access and allows for revocation of access when a device is stolen. Keypad uses a remote key server, similar to Data Tethers, but relies on the user to revoke data access, rather than environmental policies. It also has no way to keep track of copies of data, limiting its usefulness in real applications.

CleanOS is a modified version of Android that encrypts data when it is not in use, storing the key remotely in the cloud. CleanOS was built on TaintDroid, and uses its tracking system to determine where sensitive data is stored[60]. While having the same goal as Data Tethers, CleanOS offers less protection, because

stolen devices still can have active data in memory that can be exposed to an attacker.

In [53] a system for Android is proposed to control personal information by explicitly asking the user for permission to release information, each time it is released, rather than at application installation time. While this system would protect against unintentional or malicious apps releasing data, it again, offers no protection against theft. Additionally, Data Tethers would release this data, but only in an encrypted form that the attacker would be unable to use.

10.4 Code Rewriting

There are several dynamic code rewriting libraries available, including DynInst [25], Pin [42], and DynamoRIO [15]. Each of these libraries provides different programming models, instrumentation methods, and supported architectures. While these libraries offer alternatives to the DynInst library used in Data Tethers, they offer no significant advantage.

10.5 Policy Languages

There are several policy languages that have been created for the purpose of restricting how data is used. Pretschner *et.al.* [51] introduced such a language. Others include OSL and ODRL [35][33]. These languages were designed for slightly different purposes than the Data Tethers language, and cannot express Data Tethers style policies as clearly and easily as the existing language.

CHAPTER 11

Future Work

While the current Data Tethers implementation is, in our opinion, a fairly complete proof of concept system, we anticipate that substantial future work is left to be done. Some of this work involves implementing new features not in the current system, mainly due to time constraints. Others are extensions that became of interest while we were implementing Data Tethers. The third type of future work we anticipate is the constructions of new systems using the underlying data flow tracking mechanism offered by Data Tethers. In this section we will discuss this future work in detail.

11.1 Implementation Features

The first type of future work we would like to address is the addition of future features to the Data Tethers system. Most of these features were discussed during our design of the original system but were not implemented due to time or, in some cases, hardware constraints. However, in order to have a more complete and usable system, we feel that many of these features merit development.

11.1.1 TPM

The lack of a TPM hardware in our development environment was a major impediment in our development of the system. The fundamental underlying security of Data Tethers relies on being able to certify that the boot chain is all composed of trusted components. While we can issue fake attestations for the purposes of testing, and this has no real impact on performance, for a field-able system, having this piece in place is crucial.

Adding attestations would be slightly tedious, but not difficult. We would have to add our own trusted keys to the BIOS, boot loader, and operating system. We would then need to sign the boot loader, the operating system, the policy monitor, and the various modules. This would enable us to create an attestation that all software that was running was a correct version.

11.1.2 IPC

Another piece of the Data Tethers system we think should be implemented in the future is data flow tracking in the remaining IPC mechanisms. While systems like Flume have suggested mechanisms for enforcing information flow polices for IPC, the Data Tethers case is different since we are maintaining labels, not enforcing policy, and thus are interested in security elevation, rather than enforcement. We also feel this ability to track more flows between processes would enable us to implement new types of applications, for example, tracking how private user data moves through the operating system. Exactly what we would need to implement is discussed in detail in Section 6.1.3.

11.1.3 Multi-threading Support

Most applications these days are multi-threaded to utilize modern multi-core architectures. Because Data Tethers relies on watchpoints to trigger instrumentation, and because inside a safe function these watchpoints are disabled, Data Tethers cannot handle multi-threaded applications correctly. The simplest way to correct this would be to change the threading model from a shared address space, to each thread having its own address space, with all pages mapped in as shared. This would allow us to disable watchpoints for one thread, while leaving them enabled for other threads, but would have an additional overhead in thread context switching.

Watchpoint issues aside, there is an additional issue with threading that must be considered. While correct thread programming *should not* modify shared data without some sort of mutex or other shared access lock, this is not enforced in applications, but is left up to the programmer to implement correctly. While failure to synchronize is a bad programming practice that will lead to unpredictable execution, the resulting corruption of labels is a much more serious issue, and could lead to severe data leaks and would represent a major threat to security.

11.1.4 Environmental Monitors

A very limited number of environmental monitors were implemented for the Data Tethers system. While these monitors do not represent the major intellectual contribution of the work, the creation of them offers some research challenges. For example, the problem of how to securely determine and verify location is a well studied problem. As part of the Data Tethers project, we implemented

on such location based monitor, though there are several other ways it may be implemented. Additionally, there may be new, novel types of environmental monitors that offer new advantages from a security perspective.

11.1.5 Data Tethers Bypass API

In earlier sections we mention the possibility of creating applications that use policy-controlled files, but do not necessarily use the data inside them. For example archiving tools, FTP servers, database servers, etc. For these applications, there is no real reason to pay the cost of instrumentation, when they are only interested in the data as chunks, and not in the actual contents. For these types of applications, we could implement a bypass mechanism, to allow them to request raw, encrypted data. This would be relatively simple to implement, but given our focus on end user machines, rather than servers, of low priority.

11.2 Information Flow Tracking Extensions

There are several extensions to the Data Tethers information flow tracking system that we feel would be advantageous to implement. The current system, while useful because of its broad applicability to current applications, has significant issues, particularly with respect to implicit flow tracking. While this is generally a hard problem, we feel that there are several potential avenues to explore in this respect.

11.2.1 Integration With Information Flow Tracking Languages

One promising area for extending the Data Tethers system is to integrate it with an existing information flow tracking language. This would give us much stronger guarantees on flow correctness, while still leveraging the Data Tethers infrastructure. Unfortunately, there are very few to no applications written in these languages, so providing comparable performance and usability analysis would be difficult. However, this would provide the added benefit that it would provide these languages with a practical platform to be benchmarked and compared on.

11.2.2 Extended Instrumentation

In Chapter 4 we show that it is not possible with the current programming model to correctly instrument implicit flow tracking. However, it is entirely possible that we could in some way limit the programming model, as part of the instrumentation process, in such a way that we could conservatively instrument implicit flows. For example, implementing some type of dynamic type discovery system for allocated memory and enforcing type constraints on register loads and stores at run time. This would enable us to narrow down the number of possible addresses that need to be labeled in our loop example to all addresses of the type that is stored to in the loop.

11.3 Systems Based on Data Tethers

The information flow tracking system in Data Tethers has many uses beyond environmental policy enforcement. The fact that it works with existing applica-

tions makes it particularly attractive for projects that seek to extend or improve practical systems. In this section we will discuss several such projects, some of which are ongoing.

11.3.1 Linux Port

While there were many reasons for developing the original Data Tethers on OpenSolaris, it can be said without any doubts that the most common research operating system currently is Linux. Because of the many applications and systems implemented on Linux, having a Linux port of Data Tethers would be invaluable. Additionally, Linux runs on a much wider variety of hardware and form factors than OpenSolaris, allowing for many new uses of Data Tethers. We are currently working to develop a Linux port of Data Tethers. The Linux port represents a significant challenge because of the much larger instruction set, the variety of addressing modes, and the lack of a built-in software watchpoint system.

11.3.2 Semantic File Equivalence

Another project that is currently under development is to use the Data Tethers flow tracking system to determine "semantic equivalence" between files. A file is semantically equivalent to another file if it has been derived by an application from the original file. For example, a photo that has been cropped and blurred, then saved to a new file is semantically equivalent to the original. The goal of this project is to provide deduplication and recovery using this semantic equivalence understanding to regenerate files on demand.

The Data Tethers system has been modified to provide fine grained prove-

nance information, indicating how files have been derived from other files. Future modifications might allow us to track functions calls and parameters to more accurately identify the process by which the files are derived. This would allow us to store only the original files, and reconstruct derived files by playing back these functions. This would offer significant storage savings as well as a novel way to restore lost of damaged files.

11.3.3 User Data Privacy

Another extension of the Data Tethers system is to use it as a tool to provide user control of how their personal information is used after it leaves their computer. For example, a user may submit a payment via a web form, and attach a policy to their personal data allowing access by specific entities and for a limited duration. A modified version of the Data Tethers system would then be run on the server, enforcing these conditions for data access.

A prototype of this system has been implemented as part of a Master's student thesis, but more work needs to be done. Ideally, a browser plugin could be created to allow for the labeling of form data, prior to posting. This data would then be processed by a simple web server designed with Data Tethers instrumentation in mind.

11.3.4 Data Flow Tracking in Server Side Applications

The Data Tethers system was primarily designed for end user applications such as word processors and spreadsheets. There are, however, many uses for a Data Tethers type system for server applications. Data loss from servers is a very

serious problem, and a system that could guarantee that any data that left the system was properly tethered and encrypted would be extremely useful.

Unfortunately, the instrumentation done in Data Tethers would be extremely costly on server applications. While desktops use very small amounts of processor time, with bursts of high use, servers have moderate to high loads almost constantly. A 6-7 times increase in CPU time is simply not acceptable in this situation.

One solution would be to implement key server side components with information flow tracking built in. For example, a database or web server could be written in a security typed language. Another approach would be to write these same applications with Data Tethers instrumentation in mind. In this case, we might avoid conditionals using data we know may be policy-controlled, to avoid implicit flows, or we might avoid copying memory that we know will be labeled to avoid the large explicit flow penalty.

CHAPTER 12

Conclusion

12.1 Summary of the Problem

As computers get more and more mobile, the data that they require goes along with them, creating a growing problem with data loss. Solutions like disk encryption provide some security against data loss on from a lost device, but only work under certain circumstances - for example that the computer is not running when it is stolen, that the password chosen is sufficiently complex, and that they are configured to encrypt all locations where copies data could be stored, just to name a few. Full disk encryption offers no protection at against accidental loss due to user error or mis-configured software.

Leakage detection software can protect against these last two cases, but fails for many common cases such as encrypted data, and also lacks any kind of guaranteed leak detection. It also offers no protection at all for stolen or lost devices, since in most cases is can be trivially bypassed.

Information security policies offer a "perfect" solution, in the sense that we can specify very secure conditions under which data can be accessed. Unfortunately they offer no enforcement mechanism for these policies. Because of this, security policies are often ignored when the user finds them inconvenient or limiting, or

merely by error.

12.2 The Data Tethers Solution

The Data Tethers approach is to prevent accidental data loss by specifying safe environments in which data can be accessed by attaching environmental policies at the word level. When the data is not in a safe environment, we secure it with strong encryption, with keys only stored remotely. We further protect against accidental loss due to user error by ensuring that whenever secure data leaves the machine it is always encrypted.

The key component of the Data Tethers design is the information flow tracking system. This system dynamically instruments running executables on demand, inserting instructions to track sensitive data in use by the application. Through this tracking system, we propagate labels to derivative data and files or data sent over the network. This process enables us to ensure that all copies of sensitive data remain sensitive, without over-labeling.

12.3 Summary of the Data Tethers Design

The Data Tethers system is composed of three major parts. The first of these is a modified version of OpenSolaris which supports policy labels in files and network streams. The second is a policy management and enforcement system composed of the policy monitor, policy server, and policy database. The third is a dynamic instrumentation system for tracking labeled data in existing application. These three components work together to provide a complete solution for accidental

data loss.

When policy-controlled data is read in by an application, the operating system first verifies that the relevant policies are satisfied by requesting encryption keys from the policy management and enforcement subsystem. This subsystem handles requesting the policy, verifying conditions are secure, and transferring the relevant encryption keys over the network. It then hands these keys to the operating system to use in decrypting the data. The policy monitor then continues to monitor environmental conditions. When unsafe conditions are detected, it notifies the operating system to revoke access.

The third major component, the dynamic instrumentator, is also triggered by access to policy-controlled data. This application rewrites functions on demand, inserting code to track labels along with data, on a function-by-function basis. To support this, the operating system is further modified to support on demand instrumentation by including a watchpoint tracking system for labeled data, as well as interfaces for turning this system on and off. Accesses to labeled data triggers a watchpoint fault, which then results in the accessing function being rewritten.

Whenever labeled data leaves a process address space, this data is encrypted and a policy label is attached. This is handled in the operating system when the application uses a system call to write data. The exact implementation varies depending on where the data is written, and is implemented on a case-by-case basis.

We demonstrated that the while the cost of the Data Tethers system can be quite high in the worst case, in the average application the increase in wall

clock time is less than 5%. Additionally we showed that both the network and file system performance was reasonable. For user applications and end user machines, Data Tethers is an acceptable solution from a performance perspective.

12.4 Lessons Learned

The experience of designing and implementing Data Tethers offered several major lessons:

- **Modern operating systems need new designs for mobile security.** The standard operating system security model is mixed collection of mechanisms: network streams, files, pipes, etc., all use different security mechanisms. None of these mechanisms are intended to prevent data from escaping the system, but are simply focused on local user access. Leakage security is implemented by running applications as specific users with access to certain files, and hoping that these user applications do the right thing. There is no mechanism to control what applications can do with the data, aside from read and write permissions. While this was sufficient when users were working on centralized servers with a few centrally vetted and installed applications, it is not sufficient for modern mobile devices, which tend to be single user and run many user installed applications, often from unsecure sources.

- **Operating systems need centralized security.** Retrofitting the Data Tethers security additions into OpenSolaris was much more difficult than originally expected. The security mechanisms for the various ways a process

can receive data are all implemented in different parts of the operating system. There is no central "checkpoint" for all data that flows into and out of a process, besides the kernel itself. This makes it difficult to augment the security model, and provides a much larger attack surface for infiltrators than is necessary.

- **Implicit flow tracking is hard.** The fact that implicit flow tracking is difficult is a well known fact. As we showed earlier in this dissertation, in the most general case it is not possible without labeling all memory addresses. Other work has shown that it is possible under a more restrictive model[56], but given that the cost of the "best effort" tracking in Data Tethers is already high, it is very possible that implementing this more restrictive model would push the performance cost outside the acceptable range for many applications.

- **Only a small percentage of functions in a typical user application access data.** From the user application data, we saw that the amount of CPU increased about twice the base, for explicit flow tracking vs 6.4 times the base time for the micro-benchmarks, meaning we instrumented a little over 33% of the functions that executed. If we calculate this same percentage for all benchmarks, we see that for a typical user application, about 30-40% of the executed code handles tethered data. The remaining code draws GUI elements, and performed similar non-data related tasks. This was a smaller percentage than we expected, especially given that we sometimes instrument functions that do not access secure data, but simply generate too many page faults.

- **The programming model for current applications is not suitable for data flow tracking.** The standard programming model for user applications has no secure way to store labels so that a malicious or malformed application cannot tamper with them. There is also no way to relocate secure data to single pages, which would minimize label pages, allowing for simpler access detection and enforcement, and there is no support for a security context for executing code, aside from the usual user/kernel modes. Any of these extensions would make data flow tracking much more efficient and secure.

- **There is no well defined characterization of what "usual end user behavior" is.** Surprisingly, we were unable to find any studies regarding how a typical user in various environments actually uses his computer. This made deciding how to benchmark Data Tethers quite difficult. Deciding exactly how most users use spreadsheets or word processors or similar tools without any statistics to support our decision leaves the testing methodology open to questions that unfortunately no one has the answers to.

The overall lesson to be learned is that while fine-grained policy-controlled data and information flow tracking integration into an existing operating system is feasible and necessary, it would greatly benefit from better architectural support from both a hardware and an operating system perspective. This would be similar to the types of changes made to efficiently support virtual machines. We feel that if this type of support were added, that both the performance and security of systems like Data Tethers would greatly improve.

12.5 Contributions

The research behind this dissertation has made contributions in the following two areas:

- **New security model for mobile data.** The construction of the Data Tethers system showed that is is feasible to implement environmental policy based, fine-grained security as part of an existing operating system. This security model offers significantly stronger guarantees than existing mechanisms, while never being weaker. While some of these ideas have been built into experimental operating systems, Data Tethers shows that it is both possible and practical to do so in a production quality operating system, and with a large range of applications.

- **First complete integration of an existing operating system with a implicit and explicit flow tracking system for existing applications.** Data flow tracking has been implemented in many different ways, but Data Tethers is the first integration of both implicit and explicit flow tracking in an existing operating system that works with existing applications, without requiring recompilation. Given that data flow tracking is of interest in many different areas of computing, the information flow tracking portion of Data Tethers is both an interesting proof of concept for these types of systems, and also an interesting test bed for many applications of DIFT.

12.6 Final Comments

Information loss is a serious problem facing the ever-expanding world of mobile computing. Data Tethers is a system that provides protection against loss or theft of mobile devices by providing a framework for attaching and enforcing arbitrary environment policies to user data. To maintain these labels, Data Tethers incorporates a dynamic information flow tracking system to automatically combine and propagate labels inside of applications using this data. The end result is a system that is secure without placing an excessive burden on the end user.

In this dissertation we discuss in detail many of the design decisions we have made, and the issues driving these decisions. Some of these problems are based in the need for security, and some are based in the need to make the system usable. This combination of security and usability we feel is necessary for a practical security system.

The resulting Data Tethers system is both usable and secure, offering a flexible language for specifying safe environments for data access, a secure mechanism for enforcing those policies, and a flow tracking system to ensure that the policies move with the data.

References

[1] 5 reasons to look forward to danbury technology, 2007. [Online; accessed 2-January-2013].

[2] Trusted platform module. In *ISO/IEC Standard 11889-1*, 2009.

[3] Data loss prevention: Keeping your sensitve data out of the public domain, 2011. [Online; accessed 2-January-2013].

[4] Bitlocker drive encryption overview, 2012. [Online; accessed 2-January-2013].

[5] Mcafee endpoint encryption for pc and mac 6.2 administration guide, 2012. [Online; accessed 2-January-2013].

[6] Open letter to rsa customers, 2012. [Online; accessed 2-January-2013].

[7] Os x: About filevault 2, 2012. [Online; accessed 2-January-2013].

[8] Protecting data by using efs to encrypt hard drives, 2012. [Online; accessed 2-January-2013].

[9] Regions says employee 401k data lost when auditor ernst and young mailed flash drive and code key together, 2012. [Online; accessed 2-January-2013].

[10] Rsa data loss prevention suite, 2012. [Online; accessed 2-January-2013].

[11] Tcg storage security subsystem class opal version 2.00, 2012. [Online; accessed 2-January-2013].

[12] Byron Acohido. Ftc sounds alarm: data leaking onto p2p networks, 2011. [Online; accessed 22-July-2012].

[13] E. Bonelli, A. Compagnoni, , and R. Medel. SIFTAL: A typed assembly language for secure information flow analysis. *Technical report, Stevens Institute of Technology*, 2004.

[14] Dan Bowman. Report: Data breaches from unencrypted devices up 525 [Online; accessed 22-July-2012].

[15] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2000.

[16] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[17] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[18] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2011-2016. 2011.

[19] James Clause, Wanchun Li, and Ro Orso. DYTAN: A generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[20] OASIS Standards Committee. extensible access control markup language (xacml) version 3.0, 2010. [Online; accessed 22-July-2012].

[21] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, pages 133–147, 2005.

[22] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. of the 37th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2004.

[23] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 482–493, 2007.

[24] B. Demsky. Garm: Cross application data provenance and policy enforcement. In *ACM Transactions on Information Security*, 2011.

[25] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun of the ACM*, 1977.

[26] Glenn Durfee, D. K. Smetters, and Dirk Balfanz. Posture-based data protection. *Technical Report, PARC*, 2007.

[27] Petros Efstathopoulos, Maxwell Krohn, Steve Vandebogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazires, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proc. 20th ACM Symp. on Operating System Principles (SOSP*, pages 17–30, 2005.

[28] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[29] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, 1987.

[30] Roxana Geambasu, John P. John, Steven D. Gribble, Tadayoshi Kohno, and Henry M. Levy. Keypad: An auditing file system for theft-prone devices. In *Eurosys*, 2011.

[31] Google. Two-step verication., 2013. [Online; accessed 07-May-2013].

[32] The Internet Society Network Working Group. Rfc 2904, 2000. [Online; accessed 22-July-2012].

[33] W3C ODRL Group. Odrl community page, 2013. [Online; accessed 07-May-2013].

[34] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. 17th USENIX Security Symposium*, 2008.

[35] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In *Computer Security ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*, pages 531–546. Springer Berlin Heidelberg, 2007.

[36] J. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference (SPHCC)*, 1994.

[37] Apple iCloud. Find my iphone, ipad, and mac., 2013. [Online; accessed 07-May-2013].

[38] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[39] Sachin Katti, Andrey Ermolinskiy, Martin Casado, Scott Shenker, and Hari Balakrishnan. S3: Securing sensitive stuff. In *USENIX OSDI 2008 Work in Progress Report*, 2008.

[40] L. C. Lam and T. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proc. of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference (ACSAC)*, 2006.

[41] Robert Lemos. Corporate america's lost laptop epidemic, 2010. [Online; accessed 22-July-2012].

[42] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[43] Philip MacKenzie and Michael K. Reiter. Networked cryptographic devices resilient to capture. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 12–, Washington, DC, USA, 2001. IEEE Computer Society.

[44] Ricardo Medel, Adriana Compagnoni, and Eduardo Bonelli. A typed assembly language for non-interference. In *Proceedings of the 9th Italian conference on Theoretical Computer Science*, 2005.

[45] Marvin L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[46] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, pages 228–241, 1999.

[47] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16, February 2008.

[48] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commidity software. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[49] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical prevention of large-scale data leaks. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[50] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1), January 2003.

[51] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. Mechanisms for usage control. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, 2008.

[52] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting general security attacks. In *Proc. of Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[53] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proc. of the IEEE Symposium on Security and Privacy*, 2012.

[54] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[55] Kevin Sack. Patient data landed online after a series of missteps, 2011. [Online; accessed 22-July-2012].

[56] E.J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331, 2010.

[57] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, 2008.

[58] Ahren Studer and Adrian Perrig. Mobile user location-specific encryption (mule): using your office as your password. In *Proceedings of the third ACM conference on Wireless network security*, 2010.

[59] G.E. Suh, J.W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. of the 11th International*

Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS), 2004.

[60] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: Limiting mobile data exposure with idle eviction. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[61] AppArmor Project Team. Apparmor security project wiki, 2013. [Online; accessed 07-May-2013].

[62] SELinux Project Team. Selinux wiki, 2013. [Online; accessed 07-May-2013].

[63] SPARK/Ada Project Team. Spark ada, 2013. [Online; accessed 07-May-2013].

[64] Mohit Tiwari, Hassan M G Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proc. of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[65] Tore Torsteinbo. Data loss prevention systems and their weaknesses, 2012. [Online; accessed 2-January-2013].

[66] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proc. of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[67] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *14th International Symposium on High Performance Computer Architecture (HPCA-14*, 2008.

[68] Wikipedia. iphone — Wikipedia, the free encyclopedia, 2012. [Online; accessed 22-July-2012].

[69] Wikipedia. Xiaomi 2 — Wikipedia, the free encyclopedia, 2012. [Online; accessed 22-July-2012].

[70] D. Yu and N. Islam. A typed assembly language for confidentiality. *Technical report, DoCoMo USA Labs*, 2005.

[71] Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. In *European Symposium on Programming*, 2001.

[72] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazires. Making information flow explicit in HiStar. In *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 263–278. USENIX Association, 2006.

[73] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazires. Securing distributed systems with information flow control. In *Proc. of the 5th USENIX Symposium on Network Systems Design and Implementation*. USENIX Association, 2008.

[74] Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Neon: System support for derived data management. In *Proc of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2010.