Architecture for Data-Centric Security

Yu-Yuan Chen

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Electrical Engineering
Adviser: Ruby B. Lee

November 2012

Abstract

In today's computing environment, we use various applications on our various computing devices to process our data. However, we can only implicitly trust that the applications do not do anything harmful or violate our desired confidentiality policy for the data, especially when those applications are run on today's feature-rich and monolithic commodity operating systems. In this thesis, we present two approaches – with and without modifying the applications – that aim to provide data confidentiality protection *after* the data is given to an authorized recipient – a problem which we refer to as illegal secondary dissemination. We also aim for the protection of the data throughout its lifetime.

The first approach follows the school of thought of providing a *secure execution compartment* for the security-critical part of an application. We propose to use the hardware to directly protect a trusted component of an application, which in turn controls access to the protected data, on top of an untrusted operating system. We devise a methodology for *trust-partitioning* an existing application into the trusted component, leaving the rest of the application untrusted. The trusted component can be used to implement the desired confidentiality policy for our sensitive data and guarantee that the policy is enforced for the lifetime of the data. We demonstrate this first approach by showing how the difficult-to-achieve originator-controlled (ORCON) access control policy can be enforced with the real-world *vi* editor.

Our first approach essentially ties the protected data with the trusted part of the application that is protected by the hardware. However, this results in the inconvenience of having to use only a particular application to access the protected data, limiting the portability and availability of the data. Therefore, my second approach removes the applications from the trust chain and provides an application-independent *secure data compartment* that tracks and protects the data in the hardware, no matter which untrusted application or authorized recipient is given access to the data. We use the flexibility of software to interpret and translate high-level policies to low-level semantics that the hardware understands, and we use the hardware to persistently track the usage of the sensitive data and to control the output of the sensitive data from the machine. We have prototyped the architecture on the OpenSPARC processor platform and show how unmodified third-party applications can be run while various data-specific high-level policies can be enforced on the sensitive data.

My second approach leverages a technique called Dynamic Information Flow Tracking (DIFT), which has been shown to be a powerful technique for computer security, covering both integrity and confidentiality applications. However, the false-positives and false-negatives of DIFT techniques have hindered its practical adoption and usability. We take a deeper look at the practicality and usability issues of DIFT and explore various techniques to address the false positives and false negatives, arising from the undecidability of conditional branches, which is a type of implicit information flow that is particularly hard to solve dynamically. We propose various micro-architectural and hybrid software-hardware solutions using only the application binaries and show how the combination of these solutions help build a practical and usable DIFT system.

Acknowledgments

Considering that this is perhaps the longest acknowledgment section I have ever written and will ever write, writing this section is no small feat in and of itself. In fact, it takes not just one person – myself, but *everyone* around me to actually accomplish this thesis.

First and most importantly, I would like to thank my thesis adviser, Prof. Ruby B. Lee. There would not have been a thesis for you to read if not for her guidance throughout my research, my academics, and almost everything that relates to my life in Princeton. I especially would like to thank her for her understanding through my ups and downs all these years, her faith in my ability, and her constant encouragement during my repeated attempts in the paper submission process. Prof. Lee has all the ideas in the world for me to explore, and she also challenges me in the technical details to help me think critically to improve even further.

There are several other people who have also helped me in my academics and research in Princeton. Prof. Sun-Yuan Kung was my academic adviser when I first entered Princeton. Prof. Kung helped me in many ways in my first year and he also served as one of the committee members for my Final Public Oral (FPO) presentation. Dr. Youfeng Wu from Intel has given me the rare opportunity of an internship position in my first summer in Princeton, back when I had very little security knowledge. Youfeng also served as my mentor when I was awarded the Intel Fellowship, and as one of my thesis readers. I especially thank him for helping me get my Chapter 6 in shape, in addition to his insightful comments on all other chapters, and for taking his precious time off of his sabbatical to read my thesis. Prof. Niraj K. Jha and Prof. Mung Chiang gave me valuable feedbacks during my oral general exam in my second year, of which the materials constitute some parts of Chapter 4. I thank Prof. Jha for serving again as one of the committee members in my FPO presentation. Dr. David R. Safford from IBM was instrumental in helping me prepare for the materials in Chapter 4 and he proposed to experiment with the eCryptfs secure file system, which led to my Section 1.1 in this thesis. Keen W. Chan, my manager during my second Intel internship, gave me the opportunity to experiment with the latest technology and to learn the practical aspects of security. The experience has proved to be useful for my research as well. Last but not least, I would like to thank Prof. Jennifer Rexford from the Computer Science department for being my thesis reader. Like Youfeng, she also spent her time during sabbatical reading my thesis and gave me prompt and insightful feedback.

Next I would like to mention some of my co-workers, lab mates and fellow graduate students. I thank the administrative assistants, Sarah M. McGovern, Stacey Weber and Lori A. Bailey for making everything easier. Jeffrey S. Dwoskin has helped me in various ways – research advice, web master administration, and general counseling on the life of a graduate student. David Champagne, who has done an impressive thesis, was never hesitant about sharing his experiences with me. Jakub Szefer and I shared almost all of our PhD years together. I especially appreciate his help when a simple bug in my code thwarted my progress for a few days, and all the *cat-sitting* favors that I never had a chance to return. Pramod A. Jamkhedkar, a post-doc who

I only have the privilege to work with in my last year in Princeton, was my co-author in an important paper that sped up my graduation.

There is no such thing as work-and-life-balance for a PhD student, so the friends that I have made while in Princeton have played a huge factor in my Quality-of-Life (QoL). Konstantinos Aisopos and Emmanouil Koukoumidis have been with me since my first year in Princeton, and have contributed to all those unforgettable memories that last a lifetime – sky-diving, Greece, Seattle, Portland, Taiwan, just to name a few, and there surely will be many many more to come. I especially would like to thank Kostas for his understanding, compassion and advice when I was going through one of the worst periods of my life. I would like to thank Cheng-Yao Chen for driving me around and getting lost for a full day just to shop for my first car in the United States, Lan Hsu and Li-Fan Yue for all the trips and fun we had together and in the future, Arvid Wang for all the intense basketball sessions that stepped up my game, the Huang family (Yue-Kai, Vicky and Millie) for all the delicious BBQs, Taiwanese delicacies and gatherings, the Liao family (Kung-Ching, Ya-Ching and Sophie) for the jokes and fun and my first deafening Formula-1 experience in Montreal, the other Huang family (Chiao-Ti and Chi-Hua) for their kindness to host me for my last month in Princeton, the Chuah family (Jun-Wei, Michelle and Cristine) for peeking into the Singaporean life-style, Ting-Jung Lin, Qiang Peter Liu, Silvana Endler, Constantinos Kalfarentzos, Megan, Crisantos, Arun Raman and Niket Agarwal for the intramural volleyball championship.

My family has been critical in supporting me both emotionally and financially, and I wouldn't have been in Princeton in the first place if not for my family. My dad, Hsin-Fu Chen, and my step-mom, Mei-Fang Chiu, have been nothing but supportive. They are the ones who encouraged me to study abroad in the beginning, even though I did not follow their wish to become the other kind of "doctor". I thank my mother, Man-Shu Chien, for having me. My older sister, Wan-Yi Chen, together with my brother-in-law, Terence Chan, and my dear nieces, Alyssa and Helena, have shared my PhD years and given me all the laughs and family warmth whenever I need it. I hope my accomplishment in Princeton will pave a path for my younger brother and sister, Jacky and Irene, for I wish to see their happiness and success in the future. I would also like to thank my in-laws – father-in-law, Ping-Huang Wu, mother-in-law, Pao-Pei Lu, sister-in-law, Pei-Lin Wu, and brother-in-law, Tsung-Han Wu, for their confidence in me that has given me a great deal of mental support.

Lastly, I cannot say enough "thank you" to my classmate, my best friend, my soul mate and my wife, Carole-Jean Wu. There is not a single thing that culminates in this thesis that was done without her, and it could not have been done without her. I had no idea that my life would be completely changed when Carole came to ask me questions about the first homework assignment that I had in the US. The only thing that went through my head at that moment was: "How do I answer in a way that doesn't reveal that I had almost zero background in computer architecture so I don't know the answers, and that I had a hard time trying to understand everything that was taught in English?" I do not remember what I answered to her questions, but luckily we came to know each other. For all these years in Princeton, Carole has given me love and support, whenever and wherever I need them. She has also helped improve my research in every aspect, e.g., the technical details, presentations, paper submissions, etc.

With this most important section of the thesis winding down to its end, I would like to thank you – whoever is reading this thesis, no matter what your intention of reading this thesis is. Hopefully after reading this thesis, I have given you the motivation and inspiration to create something new and innovative, which someone at someday in some place will find useful and thank you in his/her acknowledgments.

To my beloved wife, Carole,
and my dear family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advent of a new computing paradigm where everything is stored on a *cloud* that can be accessed *anywhere*, *anytime*, using virtually *anything*, the importance of data protection in terms of confidentiality cannot be overstated. Nowadays, a user's sensitive data can be stored on any kind of computing platform, including desktop, laptop, tablet and smart phones. In addition, users want the data to be available and accessible wherever they are, whether at home, in the office, on the road, or in a random coffee shop. Furthermore, users do all kinds of things to their data and use different applications to process or manipulate the data. For example, you can use Microsoft Word or a simple text editor program to edit the same document. This trend has led to a great deal of digital information leakage that we see all over the news. Particularly there has been a growing incentive for leaking medical information. According to a recent study [75] in 2011, medical records are worth \$50 each on the black market, much more than Social Security numbers (\$3), credit card information (\$1.50), date of birth (\$3), or mother's maiden name (\$6). Personal medical information is worth more not only because they can be used against a person's privacy, but also because the thieves can use it for false medical claims that are a lot more lucrative. Therefore there are more incentives for an authorized recipient to leak the information, which can be considered as an insider threat.

In this thesis, we define *data-centric security* as "the enabling and enforcement of data-specific security policies". In general there are three aspects related to data-centric security, i.e., primary authorization, secondary dissemination and lifetime[1] policy enforcement, as listed in Table 1.1.

To address aspect 1, a modern computer system employs data access control both in the software and in the hardware. On the software side, the operating system mediates and controls accesses to resources, e.g., through the `login` command for user authentication and the `rwx` attributes, commonly found in the Linux operating systems, to limit read, write and execute privilege on different files. On the hardware side, modern processor architectures support data access control by allowing the software to specify the access attributes of a memory page, e.g., read-only, read-write, user-read-only or supervisor-access-only, etc. However, once the `r` (read) access

[1]The lifetime of a piece of sensitive data begins when it is created, and ends when it is deleted or declassified as non-sensitive.

Table 1.1: Three aspects of data confidentiality protection.

Aspect 1	Only authorized users and programs get access to the data (which we call *primary authorization*).
Aspect 2	Authorized users are not allowed to violate the data's confidentiality policy, e.g., sending the data to unauthorized recipients (which we call *illegal secondary dissemination by authorized recipients*).
Aspect 3	The data's confidentiality policy is enforced throughout the lifetime of the sensitive data, irrespective of the user or application accessing it, across a distributed computing environment.

is given to an application, the application can copy the data to other locations and leak out the data. Furthermore, when the data access controls are enforced by the operating system, the Trusted Computing Base (TCB)[2] is increased dramatically to include the entire operating system and its configurations.

Another commonly used technique to address the problem of primary authorization is to use encryption. Encryption helps solve part of the problem by encrypting data with a key, so only those who have the decryption key can legitimately use the data. However, since it is desirable for an authenticated and authorized user to be able to choose any application he wishes to access the data, and the application needs to have access to the decrypted plaintext to perform useful and meaningful work, it is very important to control where data goes after it is decrypted and access is given to an arbitrary application. For example, full-disk encryption can be used to encrypt all the data on the hard drive, and this protects the data while stored, but this does not protect the data after the data is decrypted and being used. We show real-world examples of how a software attacker with system-level privilege can extract the decryption keys of an encrypted file system in Section 1.1.

The data access control and encryption work well for primary authorization, but they do not address aspects 2 and 3, which occur *after* primary authorization. Current architectures fall short in providing the access control we desire *after* our sensitive information has been decrypted and displayed. For example, the Linux X server design allows any program with a Graphical User Interface (GUI) to sniff user input of other GUI programs, or capture the displayed information in other GUI programs [80].

Consider an example of illegal secondary dissemination of personal medical records, which consist of personal contact information, blood type, immunization records, psychiatric reports, etc. An emergency medical technician (EMT) on an ambulance has access to your medical history and may try to sell that information if the patient is a celebrity. This is considered as a malicious user or insider attack. These information could also be inadvertently leaked by a doctor who accidentally

[2]The TCB is the set of hardware and software components critical to a system's security.

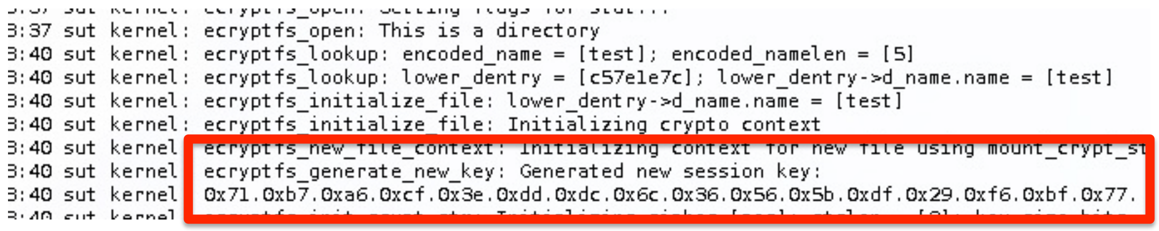
saved a copy of the record in the Universal Serial Bus (USB) drive when saving his/her vacation plans. Furthermore, the application the doctor is using to edit the vacation plan could also be leaking this information, either by a malicious malware that the doctor unknowingly downloaded or a vulnerability in the application that is exploited to leak information. This is especially true in today's computing environment, where we frequently use various applications to process our data and we implicitly trust that the applications do not do anything harmful. With the increased downloading of third-party applications from unknown sources onto smartphones, and the attractiveness of using third-party analytics programs in cloud computing environments for processing proprietary or high-value data, the protection of sensitive data processed by these applications becomes of paramount importance. A recent incident [78] where many users' entire phone books were uploaded to a remote server without their explicit consent vividly illustrates the importance of achieving data protection in terms of secondary dissemination (aspect 2) and life-time policy enforcement (aspect 3).

Since there have already been many authentication and access control techniques for primary authorization, this thesis focuses on the second and third aspects of data confidentiality protection, and the techniques proposed in this thesis can be integrated with existing primary authorization techniques. These aspects are particularly dangerous and harder to achieve, since an authorized recipient of protected information (passing the primary authorization checks) can essentially do anything he or she wants with the protected information in today's commodity systems. Even if the data is protected by encryption, the protected information will be accessible in plaintext for an authorized recipient. To illustrate the importance of protecting the plaintext and the key, we dedicate the next section to show a real-world example of how an attacker who attains system-level privilege can easily get the plaintext when the data is protected by encryption.

1.1 Importance of Protecting the Plaintext and Keys

In this section, we show how an attacker with system-level privilege can easily retrieve the plaintext and the key when data is protected by encryption. The attacker can attain system-level privilege by first infiltrating the operating system or the hypervisor, through one of many well-known privilege escalation attacks [15, 35, 69]. Once an attacker gains enough privilege in a computing device, either through software or hardware, the attacker can easily mount attacks that are otherwise impossible or difficult to achieve, e.g., access to the machine's bootloader to insert a keylogger [91] or directly accessing the data values in the main memory by compromising the operating system. In today's ubiquitous computing environment where a lot of our sensitive data is stored on our smart phones, which we may easily lose, and which is easy for an attacker to steal, the barrier for the attackers has been significantly lowered. This is partly due to the large attack surface[3] exposed by today's large monolithic

[3]The attack surface commonly refers to the set of software entities within a computer system that can be run, interacted with or controlled by an attacker.



```
3:37 sut kernel: ecryptfs_open: Setting flags for stat...
3:37 sut kernel: ecryptfs_open: This is a directory
3:40 sut kernel: ecryptfs_lookup: encoded_name = [test]; encoded_namelen = [5]
3:40 sut kernel: ecryptfs_lookup: lower_dentry = [c57e1e7c]; lower_dentry->d_name.name = [test]
3:40 sut kernel: ecryptfs_initialize_file: lower_dentry->d_name.name = [test]
3:40 sut kernel: ecryptfs_initialize_file: Initializing crypto context
3:40 sut kernel  ecryptfs_new_file_context: Initializing context for new file using mount_crypt_st
3:40 sut kernel  ecryptfs_generate_new_key: Generated new session key:
3:40 sut kernel  0x71.0xb7.0xa6.0xcf.0x3e.0xdd.0xdc.0x6c.0x36.0x56.0x5b.0xdf.0x29.0xf6.0xbf.0x77.
3:40 sut kernel
```

Figure 1.1: The target AES encryption key bit values used by the eCryptfs virtual machine, indicated by the box.

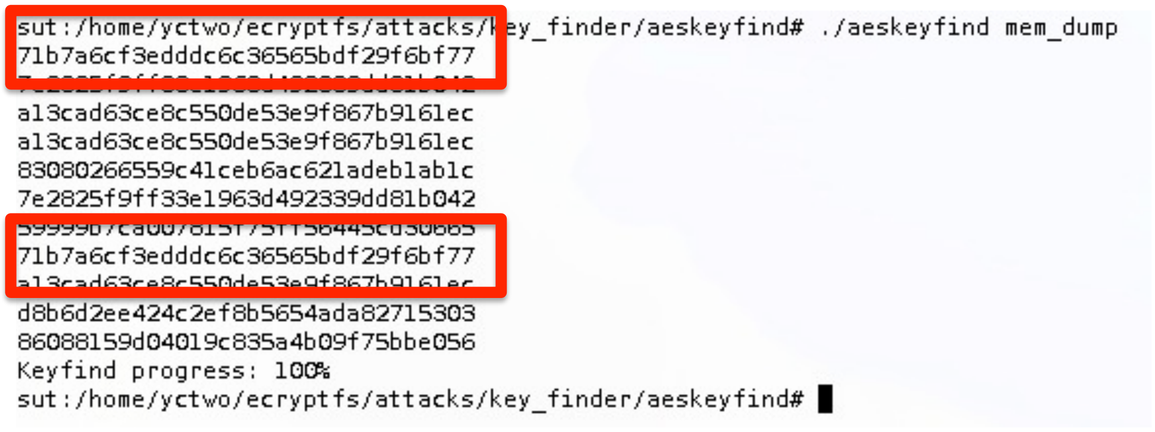
operating systems. As a result, the importance of taking into account the capabilities of attackers with system-level privilege for any new security architecture cannot be overstated. We use real-world examples of a secure file system, eCryptfs [7] and the public-private key management tool, OpenSSL [10] to illustrate this.

eCryptfs, an enterprise cryptographic file system for Linux, is a widely used data encryption technique for files in Ubuntu Linux. It encrypts data on a per-file basis, so each file has a per-file randomly-generated Advanced Encryption Standard (AES) symmetric key to encrypt and decrypt the file. The per-file AES encryption key is in turn encrypted using a key derived from the user's passphrase. In this example, the per-file AES key is the "data", which is encrypted when not in use, but is decrypted in plaintext to be used when an encrypted file is requested by the user[4]. We conduct simulated attacks on eCryptfs on our testing platform [36] to demonstrate how easy it is for an attacker with system-level privilege to retrieve the per-file encryption keys from the main memory. The attack is similar in spirit to a Cold Boot attack [51], where a physical attacker freezes the main memory module to read out the secrets stored in plaintext in the main memory. Our attack, instead of physical access, works as an attacker with system-level privilege through a compromised operating system, to achieve the same effect of the Cold Boot attack.

In our experimental setup, we have two virtual machines (VMs), one with eCryptfs running and the other simulating the attacker. While the eCryptfs VM is accessing a sensitive eCryptfs-protected file using the plaintext AES symmetric keys, the attacker controls the eCryptfs VM, e.g., by compromising the operating system, to scan the main memory, trying to find the AES encryption key used for the sensitive file. We dump the eCryptfs security log files to look for the specific key values for this particular sensitive file, as shown in Figure 1.1.

Our privileged attacker runs the `aeskeyfind` [4] tool and finds several instances of the exact key bit values used by eCryptfs to encrypt/decrypt the sensitive file, along with other bit streams that resemble an AES key in the main memory, as shown in Figure 1.2. In fact, the result shows that not only one but two copies of the exact key bit values are found in the main memory, indicating an increased probability of a successful attack.

[4]Homomorphic encryption techniques [47, 95] are proposed to operate on encrypted data. However, these techniques are still too complex to be adopted widely in practice.



```
sut:/home/yctwo/ecryptfs/attacks/key_finder/aeskeyfind# ./aeskeyfind mem_dump
71b7a6cf3edddc6c36565bdf29f6bf77
7e2825f9ff33e1963d492339dd81b042
a13cad63ce8c550de53e9f867b9161ec
a13cad63ce8c550de53e9f867b9161ec
83080266559c41ceb6ac621adeb1ab1c
7e2825f9ff33e1963d492339dd81b042
59999b7ca007815f75ff56445cd50665
71b7a6cf3edddc6c36565bdf29f6bf77
a13cad63ce8c550de53e9f867b9161ec
d8b6d2ee424c2ef8b5654ada82715303
86088159d04019c835a4b09f75bbe056
Keyfind progress: 100%
sut:/home/yctwo/ecryptfs/attacks/key_finder/aeskeyfind# █
```

Figure 1.2: Possible AES encryption key bit values found in the main memory using the `aeskeyfind` tool. The matching AES key are shown in the boxes.

Similarly, the same attack concept can be applied to finding asymmetric keys such as public/private keys in the RSA crypto algorithm. The RSA encryption equation is typically expressed as:

$$c = p^e (\text{mod } n) \tag{1.1}$$

and similarly for RSA decryption:

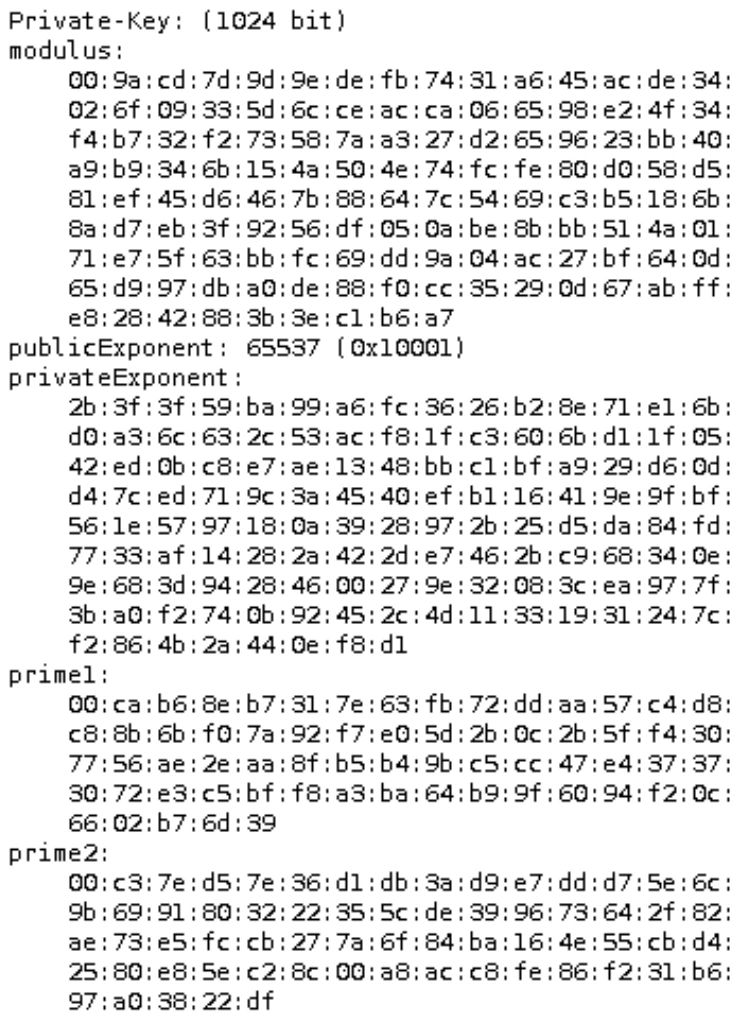
$$p = c^d (\text{mod } n) \tag{1.2}$$

where c and m denote the ciphertext and plaintext, respectively. n, the modulus, and e, the public exponent constitute the public key, whereas d is the private key. Two distinct prime numbers p and q, where $pq = n$, are also secret since they can be used to calculate the value of the private exponent d.

We use the same experimental setup described earlier, except that the victim virtual machine now runs the OpenSSL toolkit. OpenSSL is an open-source implementation of the SSL and TLS protocols that implements basic cryptographic functions including RSA public-key cryptography. We run the OpenSSL toolkit to process an RSA private key that should be kept secret since it can be used to decrypt private messages or used to perform a digital signature. Figure 1.3 shows the target RSA key components, e.g., the public/private exponents and the prime factors.

The attacker VM again controls the OpenSSL VM to scan the main memory using the `rsakeyfind` [5] tool for finding potential RSA private keys, while the OpenSSL VM is accessing the RSA private key for web transactions. The tool uses a few methods to search for the RSA key values in the memory: searching for the public modulus or the RSA key object identifier, and also searching for the identifying features of the common RSA key encoding format. Figure 1.4 shows the result of finding the exact RSA private key bits in the main memory of the OpenSSL VM.

These simple experiments show how important it is for a computer system to protect the decrypted plaintext and the key when data confidentiality is of concern and they highlight the main message that this thesis is delivering:



```
Private-Key: (1024 bit)
modulus:
    00:9a:cd:7d:9d:9e:de:fb:74:31:a6:45:ac:de:34:
    02:6f:09:33:5d:6c:ce:ac:ca:06:65:98:e2:4f:34:
    f4:b7:32:f2:73:58:7a:a3:27:d2:65:96:23:bb:40:
    a9:b9:34:6b:15:4a:50:4e:74:fc:fe:80:d0:58:d5:
    81:ef:45:d6:46:7b:88:64:7c:54:69:c3:b5:18:6b:
    8a:d7:eb:3f:92:56:df:05:0a:be:8b:bb:51:4a:01:
    71:e7:5f:63:bb:fc:69:dd:9a:04:ac:27:bf:64:0d:
    65:d9:97:db:a0:de:88:f0:cc:35:29:0d:67:ab:ff:
    e8:28:42:88:3b:3e:c1:b6:a7
publicExponent: 65537 (0x10001)
privateExponent:
    2b:3f:3f:59:ba:99:a6:fc:36:26:b2:8e:71:e1:6b:
    d0:a3:6c:63:2c:53:ac:f8:1f:c3:60:6b:d1:1f:05:
    42:ed:0b:c8:e7:ae:13:48:bb:c1:bf:a9:29:d6:0d:
    d4:7c:ed:71:9c:3a:45:40:ef:b1:16:41:9e:9f:bf:
    56:1e:57:97:18:0a:39:28:97:2b:25:d5:da:84:fd:
    77:33:af:14:28:2a:42:2d:e7:46:2b:c9:68:34:0e:
    9e:68:3d:94:28:46:00:27:9e:32:08:3c:ea:97:7f:
    3b:a0:f2:74:0b:92:45:2c:4d:11:33:19:31:24:7c:
    f2:86:4b:2a:44:0e:f8:d1
prime1:
    00:ca:b6:8e:b7:31:7e:63:fb:72:dd:aa:57:c4:d8:
    c8:8b:6b:f0:7a:92:f7:e0:5d:2b:0c:2b:5f:f4:30:
    77:56:ae:2e:aa:8f:b5:b4:9b:c5:cc:47:e4:37:37:
    30:72:e3:c5:bf:f8:a3:ba:64:b9:9f:60:94:f2:0c:
    66:02:b7:6d:39
prime2:
    00:c3:7e:d5:7e:36:d1:db:3a:d9:e7:dd:d7:5e:6c:
    9b:69:91:80:32:22:35:5c:de:39:96:73:64:2f:82:
    ae:73:e5:fc:cb:27:7a:6f:84:ba:16:4e:55:cb:d4:
    25:80:e8:5e:c2:8c:00:a8:ac:c8:fe:86:f2:31:b6:
    97:a0:38:22:df
```

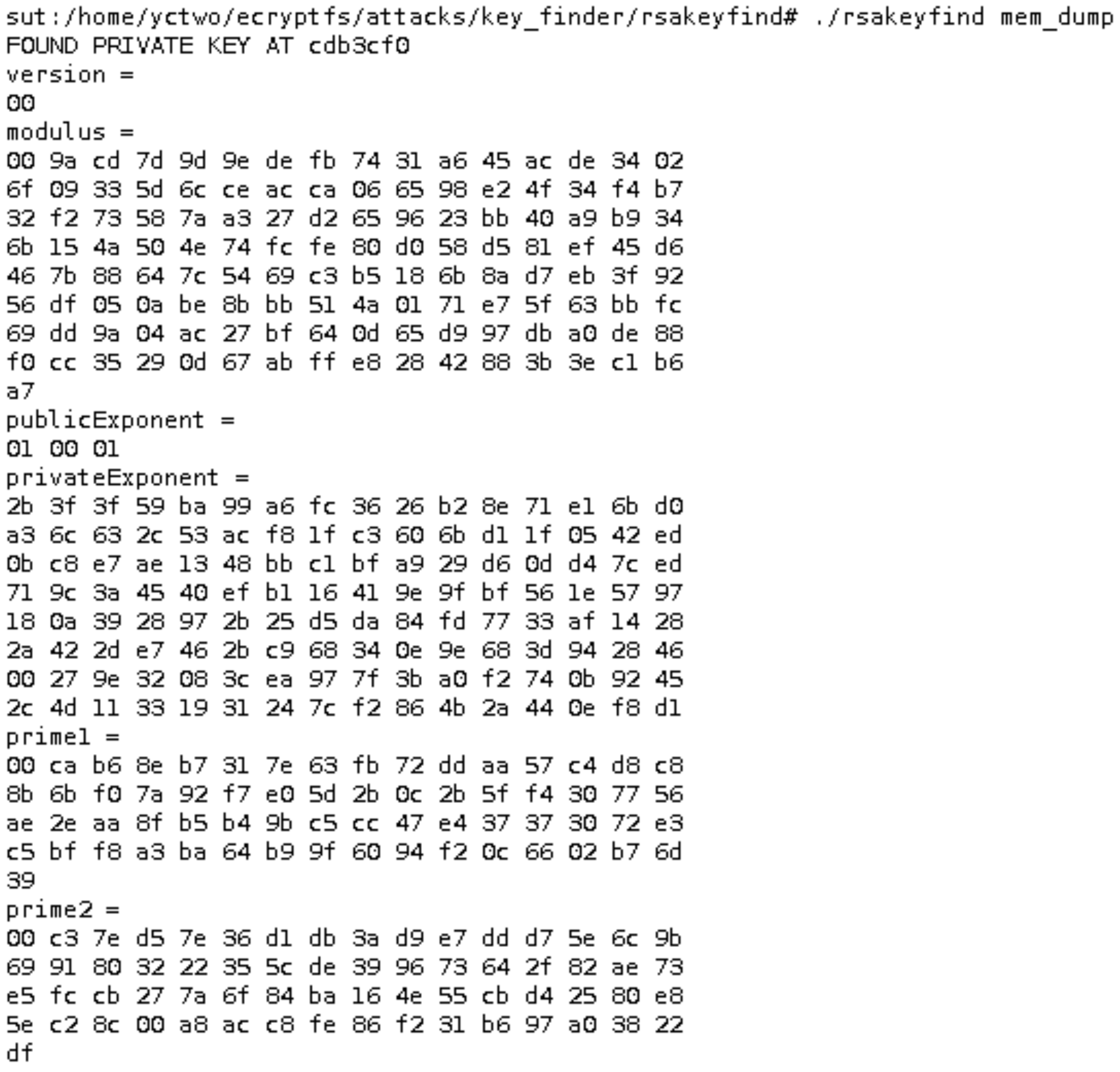
Figure 1.3: The target RSA private key components used by the OpenSSL toolkit. We show the values of the modulus (n), the public exponent (e), the private exponent (d), and the parts of the two prime numbers (p and q).

> *Encryption provides no security guarantees if the plaintext of the data and the key are not protected.*

This thesis explores different hardware-software architectural solutions to provide the desirable protections for the plaintext data and the corresponding encryption keys.

1.2 Thesis Summary

This thesis focuses on the second and third aspects of data confidentiality protection that we defined before in Table 1.1, namely the prevention of illegal secondary dissemination and the life-time policy enforcement for protected sensitive data. Specifically, the solutions proposed in this thesis are built upon the following principles:



```
sut:/home/yctwo/ecryptfs/attacks/key_finder/rsakeyfind# ./rsakeyfind mem_dump
FOUND PRIVATE KEY AT cdb3cf0
version =
00
modulus =
00 9a cd 7d 9d 9e de fb 74 31 a6 45 ac de 34 02
6f 09 33 5d 6c ce ac ca 06 65 98 e2 4f 34 f4 b7
32 f2 73 58 7a a3 27 d2 65 96 23 bb 40 a9 b9 34
6b 15 4a 50 4e 74 fc fe 80 d0 58 d5 81 ef 45 d6
46 7b 88 64 7c 54 69 c3 b5 18 6b 8a d7 eb 3f 92
56 df 05 0a be 8b bb 51 4a 01 71 e7 5f 63 bb fc
69 dd 9a 04 ac 27 bf 64 0d 65 d9 97 db a0 de 88
f0 cc 35 29 0d 67 ab ff e8 28 42 88 3b 3e c1 b6
a7
publicExponent =
01 00 01
privateExponent =
2b 3f 3f 59 ba 99 a6 fc 36 26 b2 8e 71 e1 6b d0
a3 6c 63 2c 53 ac f8 1f c3 60 6b d1 1f 05 42 ed
0b c8 e7 ae 13 48 bb c1 bf a9 29 d6 0d d4 7c ed
71 9c 3a 45 40 ef b1 16 41 9e 9f bf 56 1e 57 97
18 0a 39 28 97 2b 25 d5 da 84 fd 77 33 af 14 28
2a 42 2d e7 46 2b c9 68 34 0e 9e 68 3d 94 28 46
00 27 9e 32 08 3c ea 97 7f 3b a0 f2 74 0b 92 45
2c 4d 11 33 19 31 24 7c f2 86 4b 2a 44 0e f8 d1
prime1 =
00 ca b6 8e b7 31 7e 63 fb 72 dd aa 57 c4 d8 c8
8b 6b f0 7a 92 f7 e0 5d 2b 0c 2b 5f f4 30 77 56
ae 2e aa 8f b5 b4 9b c5 cc 47 e4 37 37 30 72 e3
c5 bf f8 a3 ba 64 b9 9f 60 94 f2 0c 66 02 b7 6d
39
prime2 =
00 c3 7e d5 7e 36 d1 db 3a d9 e7 dd d7 5e 6c 9b
69 91 80 32 22 35 5c de 39 96 73 64 2f 82 ae 73
e5 fc cb 27 7a 6f 84 ba 16 4e 55 cb d4 25 80 e8
5e c2 8c 00 a8 ac c8 fe 86 f2 31 b6 97 a0 38 22
df
```

Figure 1.4: The RSA private key components found in the main memory at address `0xCDB3CF0` on our test system.

- Using encryption and controlling the access to keys are not sufficient for protecting the confidentiality of data; the decrypted plaintext and decrypted keys must also be protected.
- To provide data confidentiality, a data-specific policy should be enforced, no matter which applications or users access the data.
- Data protection should be achieved with the least amount of trusted software entities, to minimize the overall attack surface of the system.

Building upon these principles, this thesis explores two approaches to protect sensitive data and its associated data-specific policy. The first approach, described in Chapter 4, only allows access to protected sensitive data *with* a piece of trusted application software, which executes in a secure execution environment provided by the hardware. The second approach, described in Chapter 5, enables protection of the sensitive data without any trusted application software.

In the first approach in Chapter 4, any access to the sensitive data is regulated by the trusted application software and the execution of the trusted application software is directly protected by the hardware security architecture. We leverage the Secret Protection (SP) [37, 60] architecture to provide the desired *secure execution compartment* using code integrity checking, data encryption and interrupt protection. We build the piece of trusted application software by *trust-partitioning* existing applications into a trusted component and leaving the rest of the application untrusted. We use the hardware to directly protect only the trusted component of the application software, all the while assuming that the underlying operating system is not trusted and can be compromised by an attacker with system-level privilege. Any legitimate access must be checked by and must go through the trusted application software, otherwise the access is denied and only encrypted ciphertext is accessible. This approach ties the protected data and its policy with the trusted application software such that even an authorized recipient is not allowed to abuse the data or the policy, so the confidentiality of the protected data is guaranteed by the trusted application software, whose execution environment is guaranteed to be protected by the hardware SP architecture.

The approach described in Chapter 4 requires trust-partitioning existing applications and only allows legitimate access through the trusted application software component. This partitioning has to be done for every application, making the approach harder to adopt in practice. Hence, in Chapter 5, we explore solutions where the applications remain unmodified while the access to protected data is still enforced. To achieve this, we propose a hardware-software architecture called DataSafe to provide data confidentiality protection without sacrificing practicality. DataSafe enforces the data-specific policy of each piece of protected data, instantiated within a *Secure Data compartment*, as opposed to the secure execution compartment in Chapter 4. The DataSafe software components support arbitrary high-level policies for data protection while enabling unmodified third party applications to access any DataSafe-protected data. The DataSafe hardware components uses dynamic information flow tracking (DIFT) to track the protected data and control the data output from the machine to enforce the data-specific confidentiality policy. Furthermore, the enforcement mechanism can track the data even when protected data has been transformed or encoded by untrusted applications. This approach unties the protected data and its policy from the application software, allowing an authorized recipient to use arbitrary software that he or she desires to access the data, while guaranteeing that the confidentiality policy of the data is not violated, through our runtime hardware data tracking and output control mechanisms.

The DIFT mechanism employed in Chapter 5 has traditionally been used more for integrity protection than for confidentiality protection and has not been adopted in practice due to its unacceptable levels of false-positives and false-negatives. A high false-positive would annoy the user into turning off the DIFT system and even a small false-negative implies that the DIFT system is not robust enough to track information correctly. Usually reducing one of them would drastically increase the other. To resolve these practicality issues of a DIFT system, we show in Chapter 6 both hardware and software techniques to address the false-positives and false-negatives due to implicit information flow. We propose to use a simple hardware counter, hardware save and restore mechanisms, and static analysis of application binaries, to aid the DIFT system to achieve zero false-negatives while keeping the false-positives low.

The thesis is organized as follows: Chapter 1 introduces the additional problems of data confidentiality protection and summarizes the contributions of this thesis. Chapter 2 reviews the past work in the area of data access control, secure computer architectures and information leakage. Chapter 3 formally defines the problem that this thesis solves and the threat and trust models upon which the solutions proposed in this thesis are built. Chapter 4 presents our data confidentiality protection with the access going through a piece of trusted application software which is directly protected by the hardware. Chapter 5 describes our DataSafe architecture that provides data confidentiality protection without the access going through a piece of trusted application software. Instead, the hardware tracks the protected data and enforces the data's policy through output control. Chapter 6 shows our contributions to the existing DIFT technique[5], especially for mitigating the implicit information flow problem to address the false-positive and false-negative performance of a DIFT system. Chapter 7 concludes the thesis and briefly describes future work.

[5]Note that Chapter 6 presents our solution to the DIFT sub-system of Chapter 5. We present it as a separate chapter for clarity.

Chapter 2

Past Work

As discussed in Chapter 1, important aspects of data confidentiality protection include the access control for primary authorization, the use of encryption and the protection of the plaintext and keys, and preventing the leakage of the data throughout its lifetime. In this chapter, we first look at how data access control is achieved in modern computer systems and prior proposals for enhancing it. We then discuss several security architectures that protect trusted application software to protect confidentiality, and finally we see how prior work performs information leakage detection and prevention. We defer the discussion of the related work for dynamic information flow tracking (DIFT) techniques to Chapter 6.

2.1 Data Access Control

Data access control can be done at different levels within a modern computing system, including the applications, operating system or hardware, with different techniques such as access control lists or encryption.

2.1.1 Software-Managed Access Control

At the application level, Adobe Acrobat [2] has the ability to set permissions to protect sensitive files, including viewing, printing, changing, copying or commenting. It uses simple password protection that can be broken by a brute force attack. An attacker with system-level privilege can also install a keylogger to record the password or use techniques similar to what we described in Section 1.1 to scan the memory and retrieve the plaintext data of a protected file.

To control the access to our data, modern operating systems have employed local access control for data, e.g., the `rwx` attributes commonly found in Linux operating systems – a form of Discretionary Access Control (DAC)[1]. FreeBSD incorporates a mandatory access control (MAC) security module [9] in its implementation which is able to enforce a variety of security policies on the accesses to different resources

[1]In DAC, the resource owner sets the access control rules, whereas in the Mandatory Access Control (MAC) system the rules are set by the system or by a central authority

managed by the operating system. The system implements loadable access control modules that can enforce policies according to the labels on the subjects and objects, which are specified only by the system administrator. For example, certain users can be blocked from access to some ports or sockets in the system by the MAC mechanism. Similarly, multilevel security (MLS) operating systems such as SELinux [13] use software tagging to assign and enforce the access control to sensitive data on any machine with SELinux. However, these operating system level approaches increase the TCB to include the entire operating system and its policy configurations. In this thesis, we assume the operating system is not trusted.

Several Usage control (UCON) access control models and mechanisms are proposed [73, 74, 76, 102] to extend beyond the `rwx` attributes. UCON is concerned with how data is used after access to the data has been granted – similar to the goal of this thesis. UCON defines a formal model [74] that is comprehensive to include different access control policies. For example, the UCON model can specify "A junior medical doctor can perform an operation only with the presence of a senior doctor", or "A user has to keep an advertisement window active all the time". However, the enforcement mechanisms of UCON are often implemented as a reference monitor within the operating system [73] or in multi-layers of the software hierarchy [64], i.e., the application, the windowing display sub-system and the operating system, to be able to monitor and control data usage across different system components. The nature of these access control models and mechanisms assumes a trusted operating system, whereas this thesis focuses on the architectural solutions when the operating system cannot be trusted. Nevertheless, these various access control models can be incorporated in our proposed solutions to address a wider range of application scenarios.

Data access control can also be done in a distributed manner. Secure Information Sharing Architecture Alliance (SISA) [12] is an alliance of several industry companies, aiming to provide a secure end-to-end architecture for information sharing in a distributed environment. It involves several levels of access control, e.g., physical access control, network access control, storage access control, etc., to provide extensive defense-in-depth. For example, a workstation connected to the network will display a standard login screen that uses Microsoft Active Directory for user login. Cisco's Network Admission Control (NAC) appliance would then verify that the user's device complies with the security requirements, then designates which parts of the network the user may use to access applications and content. The workstation is further protected by using behavior-based defenses to detect and block abnormal activities. The user can use Microsoft applications to collaborate and share files for which that user has been authorized. Content contained within e-mails and documents is protected using Microsoft Rights Management Services (RMS). This multi-level security approach provides defense-in-depth, and greater protection against secondary dissemination and data lifetime policy enforcement, albeit at the cost of greatly increasing the amount of software and hardware entities to be included in the TCB, e.g., everything from the physical entry gates to the software networking stacks.

2.1.2 Hardware-Managed Access Control

On the hardware side, modern processor architecture supports data access control as follows: each hardware translation lookaside buffer (TLB) entry in the processor can specify the access attributes of a page, e.g., read-only, read-write, user-read-only or supervisor-access-only, etc. This defines the access policy at the page-level granularity, with respect to whether or not the process (representing the user) has the right to read, write or execute the data on the particular page. However, current hardware access control only supports the page-level granularity and requires the operating system to configure the protection.

To address the protection granularity problem, Sentry [84] is a recent proposal on memory access control at the cache-line size granularity. In Sentry, an application can devise its own protection models within different modules, instead of the operating system doing the protection configuration. Their access control mechanism repurposes existing cache coherence states to check for access permission upon L1 cache misses to provide an extra level of `rwx` permissions at the cache-line size granularity. Sentry performs access checks when an L1 miss occurs to determine whether or not a cache line is allowed to be brought into the L1 cache. This eliminates the need for any checks if a data access hits in the L1 cache. Furthermore, the checks are only performed when an application requires fine-grained or user-level access control. The checks take place after the traditional page-sized TLB enforcement, allowing an application to set different access protections of the same data for different modules within the application.

Mondrian [98] is proposed as a memory protection scheme that completely replaces the role of the TLB by introducing its own memory protection hardware. Mondrian uses a multi-level permissions table to store the access permission information down to the granularity of words. The entries in the permissions table store an array of memory segments sorted by the segment start address. Mondrian uses a hardware permissions lookaside buffer, much like the TLB for page tables, to speed up the permission checks. The scheme provides word-level, fine-grained permission control to allow multiple domains to share data. However, both Sentry and Mondrian only support traditional `rwx`-style permissions and addresses the problem of primary authorization, the system has no control over *where* data goes after a read access is given to an application.

2.1.3 Using Encryption

Besides access control mechanisms, encryption is a commonly used technique to control access to sensitive data, by limiting the access to the encryption/decryption keys. Several commercial solutions have been proposed to address this issue using encryption, in the context of digital media and digital documents. Cryptolope [58], known as cryptographic envelopes, enables a commercial platform for the content creator and the publisher to license their content to the customers by controlling the distribution of the decryption keys. Cryptolope decouples the distribution of the data and its corresponding decryption keys. Several Digital Rights Management (DRM) solutions [3, 6, 61] focus on the copy-protection of the digital media using encryption to protect the media content. However, all these encryption solutions share a common threat model that assumes the entire box of the computing device is trusted, including both the software and the hardware. Therefore, once those assumptions are broken by attackers, e.g., by inspecting the memory using debuggers, the decryption keys are easily found and are used to evade the content protection schemes [61]. Cryptolope also assumes the same threat model as the DRM solutions – the device and the software on the device are trusted. Therefore, if an attacker can compromise the operating system or tap the memory bus, the attacker can have access to its decryption keys, using techniques described in Section 1.1.

2.2 Security Architectures for Protecting Trusted Application Software

Previous research in secure computer architectures and systems have taken the approach to protect the execution of a piece of trusted application software [25, 37, 66, 67, 85], which in turn protects the confidentiality of our data. Although these techniques do not directly address the problem related to data confidentiality, especially in terms of secondary dissemination and data lifetime policy enforcement, our proposed solutions in Chapter 4 and 5 can leverage and build upon these techniques to provide data-centric security, by protecting the execution of our proposed critical software components.

SP [60, 37, 38, 97] architecture is proposed to protect the confidentiality and integrity of the execution of a trusted software module without trusting the operating system, by introducing additional components in the processor. It uses encryption to protect the secret data accessed by the trusted software module and to protect the trusted software module from potentially malicious interrupts. The integrity of the trusted software module is also protected by the hardware code integrity checking to protect the code from malicious modifications. Our solution of protecting data confidentiality with a trusted application component (Chapter 4) builds on top of the SP architecture and we give a more detailed description of the SP architecture in Chapter 4.

Bastion [25] architecture extends the SP architecture by using a hypervisor to provide scalability for executing multiple simultaneous trusted software modules from multiple trust domains. Bastion uses hardware to protect the execution of the hypervisor, which in turn provides the protected execution environment for any number of trusted software modules concurrently. In addition, Bastion provides runtime memory protection for the hypervisor using hardware memory integrity trees to ensure that the hypervisor is not compromised even in the face of physical attackers. Both the SP and Bastion architectural solutions are able to protect sensitive data pertaining to the trusted application software.

Execute-only memory (XOM) [63] is a secure processor architecture that protects applications in an untrusted operating system environment. The protected applications running on XOM are kept in different compartments, each with its own compartment key. XOM uses a complex key management system to encrypt the memory partitions belonging to different application compartments, and it also uses hashes to protect the integrity of those memory partitions. Like the SP architecture, XOM has the ability to protect registers upon interrupts. Nevertheless, XOM has a more complex architecture and requires adding the instructions that use the hardware security mechanisms in the operating system kernel.

AEGIS [88] is another security architecture that provides memory compartments for trusted application software. AEGIS utilizes a security kernel within the operating system to provide a system where any physical or software tampering (integrity violation) can be detected and the attacker cannot obtain any information about the software or data (confidentiality violation). AEGIS uses dedicated hardware registers to define ranges of physical memory that are either unencrypted, encrypted, or encrypted and integrity-protected. They use the memory integrity tree to provide the integrity protection. These protected memory compartments are enforced by the security kernel in the operating system.

The Trusted Platform Module (TPM) [93] is an industry solution to achieve a trusted execution environment that can be used to provide password protection, disk encryption and, most importantly, a trusted boot-chain. TPM uses an external chip that can store cryptographic keys and information about the system's configuration, such that the keys can be used to sign attestation reports or encrypt data, and the system configuration information can be used to ensure the boot-time integrity of the software system, starting from the Basic Input/Output System (BIOS) code to the operating system. For example, when employing TPM protection, applications can seal a piece of sensitive information inside the TPM chip. In other words, TPM can essentially *bind* a set of files to a particular host under a specific configuration. An attacker who has compromised the system by changing any part of the software configurations would not be able to unseal the data that is protected by the TPM chip. However, TPM's protection model does not consider how the keys are used and where they are stored after they are unsealed, therefore the access control of the decrypted sensitive information is still left to the application and the decrypted symmetric keys from the TPM chip can still be obtained by an attacker by examining the memory content [51, 59], as described in Section 1.1. Hence, the protection of the sensitive data still requires the management of the keys and the protection of the decrypted plaintext.

Flicker [67] employs the newly introduced late launch instructions (Intel `SENTER` instruction or the AMD `SKINIT` instruction) in the microprocessor, together with the TPM to run a piece of trusted application software in isolation. Flicker measures the trusted application software and stores the measurements in the TPM, suspends the operating system, disables direct memory access (DMA) for the protected memory region and disables interrupts, to provide a secure execution compartment for the application.

Overshadow [29] presents a framework for protecting applications without trusting the operating system. Overshadow does not require special hardware and implements the protection mechanisms in the virtual machine monitor (VMM). Each physical memory page is viewed differently as encrypted or decrypted, depending on which software entity is requesting access. Therefore by providing the operating system with an encrypted view of the memory and the applications the normal view of its data, Overshadow can protect the privacy and integrity of application data in the face of an untrusted operating system.

2.3 Information Leakage of Untrusted Applications

In order to get useful work done on our sensitive data, any third-party application would need to have access to the *plaintext* of the data, unless fully homomorphic encryption schemes [47, 95] are adopted – and these are currently not feasible for practical adoption. As a result, the application can pass on the plaintext to unauthorized entities, if there are no proper protection mechanisms in place. The most common defenses involve isolation of the untrusted application, and information leakage prevention in the application, in the operating system and in the architecture. We first look at the techniques for isolation, and we review the past work of information leakage prevention in each level of the system. Note that these techniques are different from the information flow tracking techniques that we will discuss in detail in Chapter 6.

Isolation, or sandboxing of untrusted [45, 53, 62] applications has been widely researched to limit the scope of the damage done by the untrusted application. For example, changes made to the file system by an untrusted application are first cached and only committed after the user has verified the authenticity of those changes [62]. The technique is better suited for preventing the corruption of important system states, since any modification can be regarded as an unlawful change. However, it is difficult to detect whether or not any sensitive information is leaked, since the application can encode or transform the data in such a way that makes it unrecognizable.

TightLip [100] presents a method for detecting leakage of sensitive information by untrusted applications, with no hardware change and minor operating system changes. It uses a Doppelganger process – a process that replicates the original application process but is sandboxed to copy most of the state from the original process. The Doppelganger process is run in parallel with the original application process, but the Doppelganger process is fed with non-sensitive input whereas the original process is fed with sensitive input. The system monitors the inputs and outputs of both processes and look for divergent values to determine if the application leaks any sensitive information. TightLip is able to detect information leakage at the granularity of system-calls made by the untrusted application, but implies trusting the operating system.

Using the operating system to prevent information leakage has been proposed by HiStar [101] and Asbestos [39, 40, 96]. These two new operating systems use concepts from information flow tracking (Chapter 6), but use it at a higher-level of granularity within the system. Both operating systems label the OS-level objects, e.g., processes, to control the information flow by comparing the labels between objects in a MAC-style policy. Each object in HiStar has a label and the label specifies the security level and the declassification privilege for that object. Information flow is controlled by the HiStar kernel by comparing the labels of objects against a MAC policy defined for the system. A special label could be assigned to a thread object to signify declassification privilege, i.e., the thread could bypass the information flow restrictions. However, since the labeling is done at the granularity of the OS objects, fine-grained policy decisions within an application cannot be made. For example, an untrusted application consuming any sensitive data will not be allowed to send anything to the network even if the sent data is unclassified and unrelated to the sensitive data.

InfoShield [83] is a security architecture that restricts the access to the data to only those instructions that are allowed to access them. It uses code signing and code integrity checking to ensure that the application code has not been tampered with, and uses hardware tables to record and check pre-determined authorized instructions that are allowed to perform load/store operations on protected data, such that the protected data can only be accessed by the authorized instructions and the data can only be used in the order defined by the application. For example, a programmer can annotate the source to identify sensitive data to be protected by InfoShield. The compiler recognizes the annotation and inserts additional instructions to register the next valid program counter that is authorized to access the data in a hardware table. Every load/store instruction checks the hardware table to ensure no unauthorized instruction accesses the data. The goal of InfoShield is similar to this thesis; however, their approach is application-specific whereas this thesis focuses on data-centric protection mechanisms.

2.4 Chapter Summary

In this chapter, we reviewed three main categories of past work that are related to data confidentiality protection: (1) data access control, (2) secure architectures that protect application software for providing data confidentiality, and (3) information leakage detection and prevention for untrusted applications. A lot of these solutions have a threat model that assumes either the entire computing box is trusted, or that the operating system is trusted, which is becoming more and more unlikely due to the sheer code size of a monolithic operating system and the consideration of an insider attack. In the next chapter, we lay out the foundation, including the formal problem statement and the threat and trust model, upon which we build our solutions in later chapters.

Chapter 3

Problem Statement and Threat Model

We now define the problem of data confidentiality protection, especially with regard to the secondary dissemination and lifetime data protection of sensitive data. We then lay out the basic assumptions and describe the threat and trust models on top of which we build our data-centric protection solutions in Chapters 4 to 6.

3.1 Problem Statement

We state the problem addressed in this thesis as follows:

> *To prevent illegitimate secondary dissemination of protected data, after access to the data is given to an authorized recipient. The protected data should have a pre-defined policy that is enforced during its lifetime no matter what, when, where and how access is granted.*

For example, a policy could specify "the data can be read but not copied" (what), "access can only be granted during office hours from 9am to 5pm" (when), "access can only be granted within the office building and using the company network" (where), and "policy must be evaluated by the trusted policy handler" (how).

The above problem statement can also be defined formally as follows:

Suppose a subject $s \in S$ creates an object $o \in O$ that is to be distributed to and accessed by recipients $s' \in S$ within domain D with the following restrictions:

1. The object o cannot be accessed by subjects s' outside of domain D.
2. The object o cannot be accessed by subjects s' inside of domain D without the permission of subject s.
3. Any copy of, or transformed data derived from, the object o should include the same policy specified by s attached and enforced, across different machines.

Cross-domain object transfers are an extension of our basic model and must also enforce the object's confidentiality policy, or a higher-level policy dictated by trusted domain managers (Section 3.2.1).

3.2 Trust and Threat Models

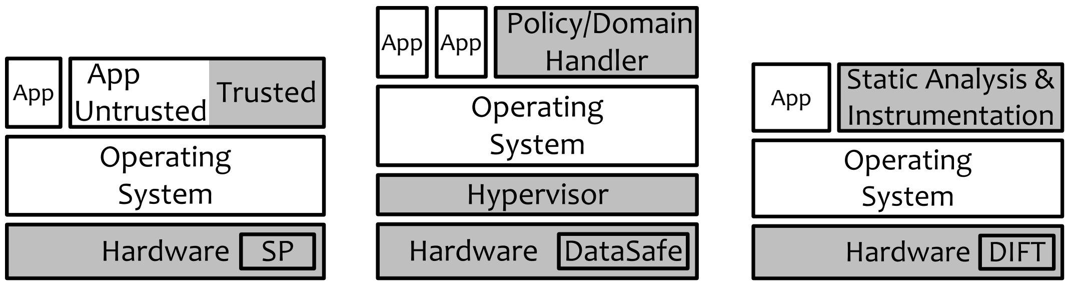
In this section we first describe the common system assumptions and threat model for Chapters 4, 5 and 6, and then we discuss their differences.

3.2.1 System Assumptions

- We assume that any protected data objects are distributed and accessed by recipients within the domain. Each domain has a domain manager that serves as the trusted authority within the domain and manages the computing devices within the domain[1].
- We assume that every recipient uses some type of computing device to access the protected data, where the hardware, including the microprocessor and the features we add in the processor in Chapter 4 and Chapter 5, is assumed to be correctly implemented and contain no exploitable vulnerabilities.
- On each of the computing devices that is enabled with the hardware data protection mechanisms, we assume that two *trusted paths* exist between the user input and the trusted microprocessor (trusted input path), and another between the microprocessor and the display output (trusted output path). Hence, the device user can be assured that the input comes directly from him/her and that what is displayed is indeed that which is processed by the microprocessor. Various techniques exist [22, 43, 71] to support a trusted input path and a trusted display, and the issue is orthogonal to the problem addressed in this thesis.
- For the low-level software running on the system, e.g., Basic Input/Output System (BIOS) and the hypervisor, we assume that secure launch or secure boot technology is employed to launch the system software and the hypervisor to establish a chain of trust to ensure boot-time and load-time integrity (e.g., Bastion [25], TrustVisor [66] or TPM [93]) for the system-level software. These secure booting issues are orthogonal to the work in this thesis and can be used in conjunction with our proposed data protection techniques.
- We assume that the authorized recipients within a domain are authenticated using standard authentication mechanisms such as passphrases, private key tokens or biometrics, through the trusted I/O paths mentioned above.
- We assume that any individual possessing the appropriate private key is legitimate and that the presence of a particular private key is enough for authentication purposes. Secure user authentication is orthogonal to our solutions and hence is not in scope for this thesis.

The different system assumptions between Chapters 4, 5 and 6 are as follows (also illustrated in Figure 3.1). In Chapter 4, we divide the application program into a *trusted* and an *untrusted* part, where the trusted part is guaranteed to perform the desired functions and any tampering with the trusted part will be detected, by means of our hardware protection mechanisms. However, the adversary can exploit the vulnerabilities in the untrusted part to perform malicious activities. In Chapters 5 and 6,

[1]Cross-domain data distribution can be achieved through key exchanges between the domain managers. We discuss the details of cross-domain data distribution in later chapters.



(a) Chapter 4. (b) Chapter 5. (c) Chapter 6.

Figure 3.1: The differences between the Trusted Computing Base (TCB) of Chapters 4, 5 and 6. Gray components constitute the TCB, whereas white components are untrusted and can be exploited by attackers. The hardware blocks SP, DataSafe and DIFT are explained in detail in Chapters 4, 5 and 6, respectively.

we relax the assumption to consider the entire application program to be untrusted. The application-independent DataSafe software components, i.e., the policy/domain handler and the hypervisor, in Chapter 5 are assumed to be trusted. The static analysis and instrumentation software components in Chapter 6 are assumed to be trusted. No hypervisor is necessary in both Chapters 4 and 6 since the operating system directly runs on top of the hardware, but if a hypervisor is in the system, it is assumed to be trusted. Compared to the commodity operating system, hypervisors are usually smaller in terms of size and complexity, and therefore expose a smaller attack surface that is easier to be verified to be secure. Figure 3.1 summarizes the differences between the Trusted Computing Base (TCB) of the three chapters.

3.2.2 Threat Model

The main goal of the adversary is to steal or leak out sensitive information that an authorized recipient is allowed to access. An authorized recipient may also be an adversary, in which case it is an insider attack. Adversaries can exploit the vulnerabilities within the untrusted operating system to leak the protected data. In this thesis, we do not consider the following threats:

1. Outright malicious software applications, e.g., viruses or worms.
2. Hardware Trojans.
3. Runtime data integrity protections as well as availability concerns, such as denial-of-service attacks, are orthogonal to the scope of this thesis. Dynamic integrity protections can be achieved by incorporating memory integrity trees [25, 46, 79] into the solutions proposed in this thesis.
4. Out-of-band attacks, such as taking a photo of the screen, or human memory.
5. Data inference attacks.
6. Covert or side channels, such as timing or storage channels.

7. Hardware attacks, such as the memory remanence attack [17, 50, 51] or chip re-writing attacks [17][2]. These hardware attacks are already addressed by the SP [38, 37, 60, 97] or Bastion [25] architecture and can be readily incorporated with the solutions proposed in this thesis.

3.3 Chapter Summary

This chapter first states the definition of the problem that this thesis is solving – illegal secondary dissemination of protected data by authorized recipients, and then lays out the fundamental assumptions and threats that our solutions proposed in Chapters 4, 5 and 6 are based upon. In the next chapter, we go into the details of our first solution – Policy-Protected Data *with* Trusted Application Software, which leverages the hardware protection provided by the SP architecture to build a piece of trusted software module that controls all accesses to the protected data to ensure data confidentiality.

[2]Chip re-writing attacks use microscopes or probing needles to overwrite the contents stored in a Read-Only Memory (ROM) or an Electrically Erasable Programmable Read-Only Memory (EEPROM).

Chapter 4

Policy-Protected Data *with* Trusted Application Software

Traditional access control to protect data on a computer system is usually implemented by the operating system (OS). However, if the OS is compromised, the access control policy enforcement can also be compromised. Therefore, applications running on top of the OS need some way to protect secret or sensitive information, in spite of a compromised OS. Given today's large and complex operating systems, and the large number of security breaches found in commodity operating systems, it is unwise to consider a commodity operating system as part of the Trusted Computing Base (TCB). Hence, in this chapter, we examine how data can be protected within an application to provide a flexible mechanism to achieve application-level access control, without trusting the operating system.

We make the following contributions in this chapter:

- implementing a distributed access control policy at the application level. We pick a policy that is known to be very difficult to enforce, e.g., originator-controlled (ORCON) [49, 65] policy, and
- developing a methodology for *trust-partitioning* of an application.

We define trust-partitioning to mean the partitioning of an application into a trusted software component for accessing protected data, while leaving the rest of the application untouched and untrusted. The trusted component is then the only way by which the rest of the application can access the protected data.

Some materials in this chapter has been published in [31].

4.1 Background

To be able to address illegal secondary dissemination and lifetime data-specific policy enforcement, one approach is to limit *which* software entity has access to the plaintext of the sensitive data. In other words, if we only allow one trusted application software to have access to the plaintext data, and we require all access requests for the data to go through this piece of trusted application software, then we can guarantee that the data-specific policy enforced by the trusted application software cannot be violated.

Therefore, the trusted application software serves as a gateway to the sensitive data, and we need to provide a *secure execution compartment* for the piece of trusted application software, to ensure that the trusted application cannot be compromised by an attacker, especially when running on an untrusted operating system.

In this chapter, we propose the following solution to data confidentiality protection:

> *A small, verifiable and trusted application module that enforces the data's policy with direct hardware protection that cannot be bypassed or manipulated by the operating system.*

Implementing the data protections in the application-space removes the dependency on the operating system and adds the flexibility of incorporating different policies, but the application must be protected from the potentially corrupted operating system. To achieve this, a secure execution compartment for the trusted application module has the requirements below:

Requirement 1: The decryption key of the sensitive data has to be protected. The decryption key can only be accessed by the trusted application module.

Requirement 2: The integrity of the trusted application module has to be protected, such that an attacker cannot modify the code of the module to perform malicious activities.

Requirement 3: The sensitive data, when decrypted in plaintext, can only be accessed by the trusted application module.

Requirement 4: Any sensitive data used by the trusted application module has to be protected during runtime, even when the execution of the module is interrupted.

Requirement 5: The trusted application module should be as small as possible to reduce the attack surface and to allow for easier security verification.

Requirement 6: The integrity of the policy of the sensitive data has to be protected.

Several of the security architectures described in Section 2.2 provide such a secure execution compartment. Our solution architecture [31] builds on top of the Secret Protection (SP) architecture [37, 60]. SP presents a minimal addition to the microprocessor, which provides a secure execution compartment for a Trusted Software Module (TSM) that meets all the requirements described above.

In SP, the protection of a TSM is proposed; however, the design of the critical components of a TSM, and the question of how to turn an existing application into a trusted part (TSM) and an untrusted part, was left unanswered in the original papers [37, 60]. We address these issues in this chapter. Specifically we show how to provide protection of existing applications by modifying them to incorporate a TSM that is directly protected by the hardware to prevent any unauthorized information leakage. We also design a generic TSM structure for general policy usage. We implement a proof-of-concept (originator-controlled) ORCON-like access control policy for protected data that is designed to be enforced in a distributed manner, in a popular Unix editor *vi*.

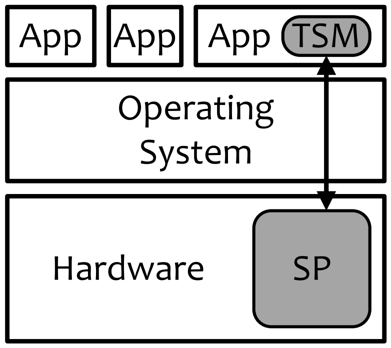


Figure 4.1: High-level view of our proposed solution. The white parts are the commodity computer system (untrusted) and the gray parts are the trusted part of an application (TSM) and the SP hardware components.

Consider the case where a secret document is to be distributed to selected recipients of different clearance levels, while the content of the original document cannot be modified. Further, the dissemination of the content has to be approved by the content creator. This policy is the Originator-Controlled (ORCON) [49, 65] policy. It is proposed to address such a scenario. Since the access control point of the policy is neither entirely centralized (policy dictated by the originator) nor entirely distributed (policy enforced for all entities), it cannot be directly solved by applying Mandatory Access Control (MAC) or Discretionary Access Control (DAC), which are typically used in traditional systems. In this chapter, we use a ORCON-like policy to show the effectiveness of protecting data confidentiality through the use of the Trusted Software Module, which is partitioned from existing applications and directly protected by the SP hardware. In the next section, we describe the baseline SP architecture, and the secure execution compartment that it provides for the TSM.

4.2 Baseline Architecture

In our solution, the trusted computing base (TCB) consists of a combination of a trusted part of the application software and the additions to the microprocessor hardware, which build upon the Secret Protection (SP) [37, 60] architecture to provide direct hardware protection of the application, as shown in Figure 4.1. SP Architecture was first proposed [60] to protect the user's secret keys (*user mode*). It also has later variants which protect a remote authority's and third parties' secret keys (*authority mode*) [37], sensitive data on sensor nodes (*sensor mode*) [38] and sensitive data in embedded systems (*embedded mode*) [97]. Our solution builds upon the authority mode SP [37]. We describe the fundamental ideas of the SP architecture in this section. Note that the original SP architecture has a threat model that assumes physical attackers, whereas this thesis does not consider physical attacks, so we describe only the relevant SP architecture, excluding the parts that address physical attacks.

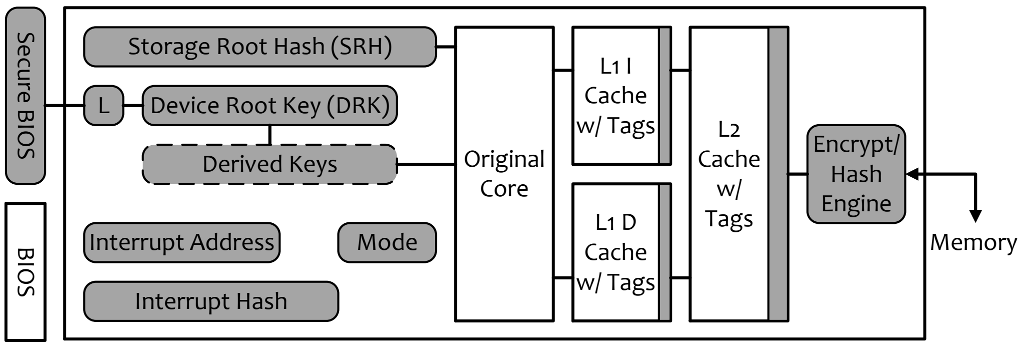


Figure 4.2: The internal components of a SP processor [37]. The derived keys block is enclosed in dashed line since it is not a real hardware component, but to show that the original core does not directly access the value of the DRK.

The SP architecture consists of the Trusted Software Module (TSM) in the user-level application and the SP hardware in the microprocessor chip, as shown in Figure 4.1. The main objective of the SP architecture is to use the SP hardware extensions to directly protect the confidentiality and integrity of the execution of the TSM, and the SP architecture achieves these by ensuring the following:

1. Each TSM is bound to a particular SP hardware and can only be executed on that SP hardware. Furthermore, the integrity of the TSM code is continuously checked when the TSM is executing.
2. The sensitive data of the TSM can only be accessed in plaintext by the TSM, not by any other software entity, and any illegal changes to the TSM data can be detected by the SP hardware.

There are two *hardware trust anchors* in the SP microprocessor chip (Figure 4.2) that enable the above protections for the TSM:

1. the Device Root Key (DRK) to bind the TSM, to protect the integrity of the TSM code and to protect the confidentiality of the TSM data in the memory and in permanent storage, and
2. the Storage Root Hash (SRH), to protect the confidentiality and integrity of the TSM data in the permanent storage.

The DRK has a few distinct properties:

1. it is unique for each processor chip,
2. it never leaves the chip and cannot be read or written by any software, and
3. the only software that can use the DRK is the TSM, via a special instruction that can derive a new key from the DRK given nonces and/or constants. The use of the DRK by the TSM is restricted to only key derivation purposes; in other words, even the TSM does not have direct read access to the DRK and thus cannot leak out the DRK.

The SRH securely stores the root hash of a secure user-defined storage structure (whether on disk or on-line storage). The SRH is accessible only to the TSM. Other software cannot read or write the SRH, including the operating system. Any sensitive data can be stored in the user-defined secure storage, and hence their integrity

(Requirements 1 and 6 on Page 22) are protected by the SRH. Furthermore, they are also encrypted by a key derived from the DRK while in this secure storage, so their confidentiality is protected as well (Requirement 1).

To ensure the TSM binding and to protect the runtime execution of the TSM, SP employs two hardware techniques to protect the TSM's code and data: *Code Integrity Checking (CIC)* and *Concealed Execution Mode (CEM)*. Hardware Code Integrity Checking ensures the integrity of the TSM code while executing. This ensures Requirement 2 (Page 22) for the secure execution compartment of the TSM. Each instruction cache line embeds a message authentication code (MAC)[1], with the DRK as the key. The MAC is verified before the instruction cache line is brought on-chip. The verified instruction cache lines belonging to the TSM are tagged in the on-chip caches. A TSM is bound to a particular SP-enabled machine with a particular DRK and cannot be executed on any other machine with a different DRK.

The generation and insertion of these MAC values for the TSM code are done in a two-step process. First, the compiler for the TSM takes two inputs, the TSM code and the target machine cache line size, to compile the binary for the TSM such that the compiler inserts nop instructions to reserve the byte locations at the end of the cache line, where the MAC value for the instructions in the beginning of cache line will occupy. The compiler is responsible for calculating the correct branch or jump targets accounting for the nops. At the second stage, the compiled TSM binary is transferred to the trusted depot, where the trusted authority uses a binary re-writing tool that takes the TSM binary and the DRK, to calculate the MAC values and replace the nop instructions with the correct MAC values in the TSM binary.

Hardware Concealed Execution Mode (CEM) protects the TSM's data while it is executing, to guarantee confidentiality and integrity of any temporary data that the TSM uses, whether this is in on-chip registers or caches, or evicted to off-chip memory. A Mode register is used to indicate whether or not the system is operating in the Concealed Execution Mode (CEM). CEM is triggered by a special instruction. The hardware begins checking the code integrity and all tagged protected data are accessible using special load and store instructions while the system is in CEM. All data cache lines containing protected data are tagged while stored in the on-chip caches. They are encrypted and hashed when evicted from the microprocessor chip. Access to tagged protected data while not in CEM will trigger a violation, ensuring Requirement 3 (Page 22) of a secure execution compartment. During interrupt handling, the contents of general registers are encrypted using a DRK-derived key and a hash is calculated and stored in the Interrupt Hash register, so that the operating system cannot observe or modify the register values without being detected, fulfilling Requirement 4 (Page 22). Furthermore, the interrupt return address is stored in the Interrupt Address register to be protected from a potentially corrupted OS.

In addition to the protection of the TSM's execution, SP architecture uses a Secure BIOS to initialize the system with a unique DRK and a lock (L) bit is used to prevent the DRK from being modified by a malicious party. Only the authority who knows the DRK can unlock the DRK and re-initialize the system with a different DRK.

[1]A MAC, also called a keyed hash, is a short piece of information used to authenticate a message.

SP also incorporates a hardware encryption and hashing engine to accelerate the automatic encryption (or decryption) and hash generation (or verification), reducing cryptographic overhead to the infrequent cache-miss handling of the last level of on-chip caches.

4.3 Application-Dependent Data Protection Architecture

Having established the baseline architecture that protects the execution of a Trusted Software Module, which is a piece of application, we now investigate two critical issues that are not addressed by the SP architecture alone:

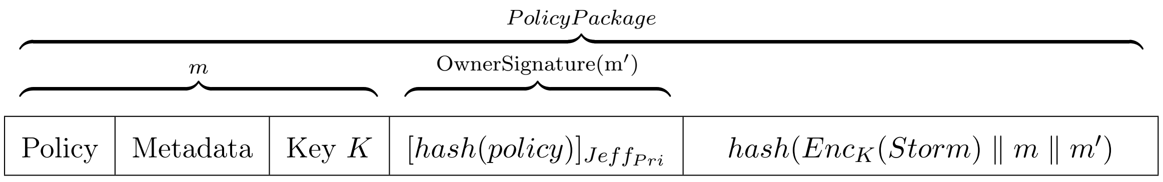
1. how we can construct or carve out a part of an existing application to become the TSM, i.e., trust partitioning the application, and
2. what should the TSM components be to enable enforcing an application-level distributed access control policy on confidential data.

Before we go into the details of the TSM components, we first look at the overall operations of the distributed access control mechanism and the structure of the protected data.

4.3.1 Summary of Access to Protected Data

We walk through a simple usage example to show how our overall architecture protects and enforces the access control of the protected data in a distributed manner.

1. The data owner creates the document containing sensitive data using any application he/she chooses.
2. The data owner dictates the policy he/she would like to enforce, e.g., *who* has *what* access to the data.
3. The data owner runs an editor application which contains the TSM, to turn the document into a protected document, which involves the following steps:
   (a) The TSM first randomly generates a new symmetric key.
   (b) The TSM encrypts the data using the generated key and erases the plaintext.
   (c) The TSM calculates the hash of the policy and asks the content creator to sign the hash.
   (d) The TSM computes a hash over the encrypted document, policy, metadata and the key, and stores them in a package called the policy package in the secure storage protected by the SP hardware.
4. The data owner can now distribute the encrypted document to all recipients he/she desires.
5. The TSM on the data owner's device encrypts the policy package using a symmetric communication key derived from the DRK and sends the encrypted policy package to the domain manager. (An alternative implementation can make use of the group encryption technique [56], which we discuss in Section 4.3.5.)


$PolicyPackage$
m
OwnerSignature(m′)
Policy
Metadata
Key K
$[hash(policy)]_{Jeff_{Pri}}$
$hash(Enc_K(Storm) \parallel m \parallel m')$

```
Policy
    Data Owner: Jeff
    Expiration date: 01012016
    Alice,  read,    -
    Bob,    read,    append
    ⋮
Metadata
    Filename storm
    Size 60KB
```

Figure 4.3: The policy package for an example protected document named *storm*. $Jeff_{Pri}$ represents the private signing key of Jeff. $\parallel$ denotes concatenation.

6. The TSM of the recipient's device requests the domain manager for the policy package. The domain manager re-encrypts the policy package using a symmetric communication key and sends the encrypted policy package to the recipient TSM (Figure 4.5). The recipient TSM securely stores the policy package in the SP-enabled computer's secure storage.
7. The TSM on the recipient's device authenticates the recipient and checks the policy before granting access to the contents of the protected document.

4.3.2 Policy-Protected Data

Each protected data package contains the data itself and a policy package that specifies the policy associated with the data and other critical information associated with the data. The policy and the data should be logically tied together. Whether or not they are physically separated is implementation-dependent. We tie together the policy and the data by a cryptographic hash. The policy package is stored and protected in the secure storage which is protected by the SP hardware. This ensures that only the TSM can legitimately access or modify the stored policies in the secure storage. Any illegitimate modifications to the stored policies will be caught by the TSM when calculating the hash value of the secure storage and comparing the hash with the on-chip Storage Root Hash (SRH) – see Requirement 6 on Page 22.

The sensitive data is protected by encrypting the document with a randomly-generated key that is stored in the policy package in the secure storage. Since the document is encrypted, it can be safely stored in any public storage without additional access control protection. The key to decrypt it is bound by the policy and the policy is enforced by the TSM. In other words, the TSM always controls the access to the decryption keys. In addition to the policy and the key, we store other pertinent information of the protected document in the policy package. Figure 4.3 shows the internal structure of an example policy package in the secure storage. The package contains a policy dictated by the data owner, metadata pertaining to the protected data, a randomly generated key to encrypt/decrypt the data, a signature of the policy signed by the owner and a hash of all the above items generated by the TSM. This policy package is stored in SP's secure storage which is encrypted by a key derived from the DRK. Hence, the confidentiality and the integrity of both the decryption key K, and the owner's policy, is protected by hardware in the secure storage (fulfilling Requirements 1 and 6 on Page 22).

4.3.3 TSM Architecture

The TSM acts as a gateway to the protected data – only legitimate access to the plaintext data is allowed through the TSM. All other accesses will either be denied by the TSM or only the ciphertext can be accessed. Before the user is allowed access to the contents in the document, the TSM first checks the integrity of the encrypted document and the policy package, to make sure that they have not been tampered with. The TSM brings in to the memory the encrypted policy package stored in the secure storage using normal load instructions, and then reads the policy package using special secure load instructions (Section 4.2). If the data cache lines for the policy package was in the on-chip caches, a secure load to these cache lines would trigger an exception, causing the cache lines to be evicted out to the memory and brought back in again through the hardware crypto engine to be decrypted, so the TSM can check the integrity of the plaintext policy. Then the TSM checks if the policy allows the particular recipient access to the contents of the document. After all checks have successfully passed, the TSM brings in the data file, encrypted under key K, to the memory. The TSM sets aside a region of memory that is to be used as the TSM buffer, decrypts the data file using the key K, and writes the decrypted data file into the TSM buffer using secure store instructions. The data file, although decrypted, gets re-encrypted automatically by the crypto engine with a DRK-derived key when the cache lines of the data file get evicted to the memory by secure stores, so no plaintext data ever appears in the memory. When the TSM gets a request to read the plaintext data, the TSM uses secure load instructions to read from the TSM buffer and sends the plaintext data through the trusted display link to display the contents to the authenticated recipient. The temporary TSM buffer is a region of memory that is accessible only through special load and store instructions during CEM and hence is only accessible to the TSM (Section 4.2). When new data are appended, the TSM is responsible for updating the policy package and the tie between the policy and data, to make them consistent. Distribution of the new file and the new policy package is required for other recipients to have access to any newly appended data.

Figure 4.4 shows a general structure of the TSM consisting of several modules that perform different functionalities required by the TSM. The TSM is constructed

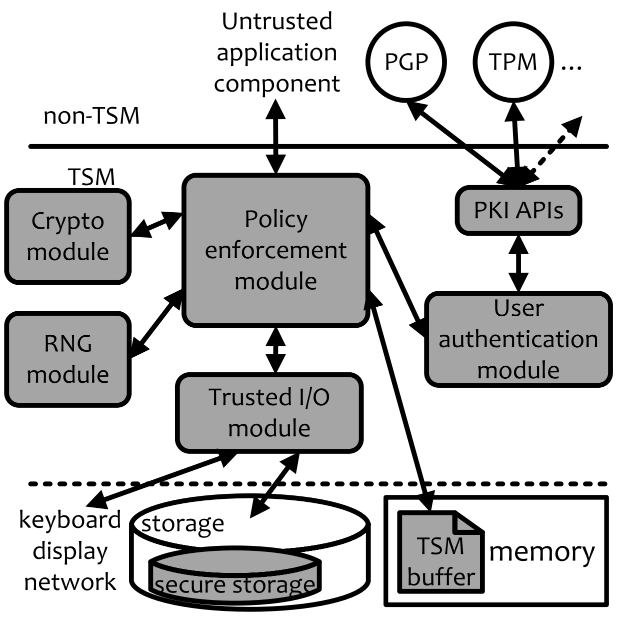


Figure 4.4: TSM architecture. The trusted (gray) parts of the system are the TSM and the SP-protected secure memory and secure storage.

in such a way that avoids being limited to a specific application and a specific access control policy.

A trusted I/O module serves as the gateway for the TSM to securely receive user input, to securely display output or to connect with other TSMs. A crypto module that implements symmetric key encryption/decryption, asymmetric key encryption/decryption and cryptographic hash functions, and a random number generation (RNG) are included in the TSM, so that the TSM does not need to depend on the operating system or other libraries for these security-critical functions. The core of the TSM is the policy enforcement module that interacts with the TSM buffer and interprets the policy stored in the secure storage to mediate the I/O of the TSM. The policy enforcement module can be tailored to implement any desired data confidentiality policy within a domain, so only one TSM for an application is needed to access any protected data object within a domain. A user authentication module, along with a set of PKI interfaces is included in the TSM to take care of the user authentication required to guarantee that the owner of the public/private key pair specified in the policy is correctly authenticated. We describe the issue of user authentication next.

4.3.4 User Authentication

User authentication is a difficult problem for the TSM, since we cannot rely on the operating system for existing user authentication mechanisms. To simplify the design of the TSM and not burden it with complex user authentication functions, we propose a public/private key authentication solution. We build a generic Application Programming Interface (API) that can interact with and make use of different pub-

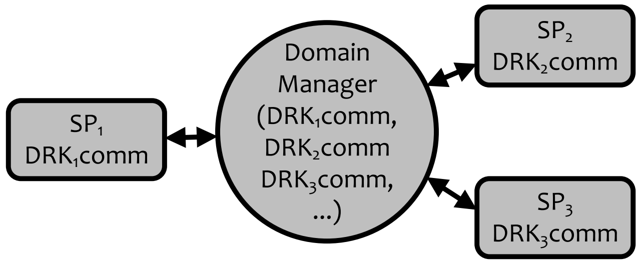


Figure 4.5: The pair-wise communication between the domain manager and the individual TSMs on different SP devices. Each SP device has its own unique DRK_i and $DRK_i comm$ denotes the symmetric communication key derived from the DRK to encrypt the communication data.

lic/private key applications, e.g., OpenPGP or GnuPG, which manage users' private keys. We outline below the protocol used by the TSM to authenticate a user utilizing other PKI applications.

When invoked by the user to read a policy-protected document, the TSM prompts the user for identity, for example, *Alice*. The TSM reads the corresponding policy in the secure storage to locate Alice's public key, $Alice_{Pub}$. The TSM calls the RNG module to generate a new random number and uses $Alice_{Pub}$ to encrypt the random number as a challenge. The TSM sends the random challenge to the PKI application through the PKI interface and asks it to decrypt the random challenge.

The PKI application authenticates the user via its normal mechanisms, e.g., a passphrase or password. The PKI application returns the decrypted challenge to the TSM. The TSM checks for the validity of the random challenge to determine if the user has been successfully authenticated.

Theoretically, the whole PKI application could be included in the TSM, since it is a security-critical function. As the PKI applications are not included in the TSM, they could also be compromised and provide fake authentication responses. However, the security-critical keys that are used to decrypt the document, and the plaintext of the document, are never released outside the TSM. Hence, even if an impostor is incorrectly allowed access to the protected data, the data never leaves the TSM and the data's confidentiality policy will not be violated.

4.3.5 Group Encryption and Trust Groups

According to the description in Section 4.3.1, each TSM must request the policy package from the domain manager. Since an SP device only stores a symmetric Device Root Key (DRK), which is known only by the SP device and the domain manager, the policy package is encrypted using a pair-wise symmetric communication key that is derived from the DRK, between any TSM and the domain manager, as shown in Figure 4.5. This means that the first time each protected data is accessed by any TSM on any machine, the domain manager needs to re-encrypt the policy package for that TSM, and therefore the domain manager may become the bottleneck if the communication channel is down. To address this issue, an alternative implementation for the policy-protected data uses group encryption [56] for distributing the protected policy package to the authorized recipients.

Group encryption is the dual of the well-known group signature scheme [19, 27, 28]. In a group signature scheme, a member of a group can anonymously sign a message on behalf of the group, without revealing his/her identity. In a group encryption scheme, the sender can encrypt a piece of data and later convince a verifier that it can be decrypted by the members of a group without revealing the identity of the recipient. The authority in both cases is the only entity that can reveal the identity of the signer in the group signature scheme or the recipient of the group encryption scheme. In a group encryption scheme, a group has one *group encryption key* and multiple *group decryption keys* associated with it. The group encryption key is public and is used to encrypt messages, while the group decryption keys are private.

In order for an originator to securely distribute the policy package to the authorized recipients using group encryption, he/she first defines a group which at least consists of all authorized recipients. Then instead of encrypting the policy package with the domain manager's public key, the TSM on the data owner's machine encrypts the policy package using the group encryption key and sends the policy package directly to the authorized recipients who are in the same group and the TSMs on those recipients' machines can decrypt the policy package using their group decryption keys.

In SP [37], a trusted authority installs all TSMs and knows the DRKs of all the SP devices. The domain manager serves as this trusted authority and could also be the authority in the group encryption scheme[2]. Under the group encryption scheme, the domain manager that initializes and installs the TSMs creates a group that includes all SP devices, and assigns each SP hardware a unique *group decryption key*, while publishing the *group encryption key* for that group, such that in the secure storage of each SP device a pair of group encryption and decryption keys is stored and tied to the particular SP hardware. Therefore the data owner can be assured that the policy package can only be decrypted by SP-enabled devices in the same group. For simplicity, we assume that all SP-enabled devices are in the same group, although different groups of SP-enabled devices can be established.

4.4 Trust-partitioning an application

In this section, we take a concrete example of an editor application to see how we can partition an application into a trusted TSM part and an untrusted part. We chose *vi* [14], one of the most common text editors in the Unix operating system as our proof-of-concept application. Our methodology can also be applied to other applications. A simplified view of the *vi* program is shown in Figure 4.6, where the program takes the input files into its internal buffers, manipulates the data within the buffer through the commands from the user, and writes the data of the buffer to the output file.

[2]Note that the authority that governs the SP-enabled devices and the trust groups need not be the same as the certificate authority in the PKI systems for user authentication.

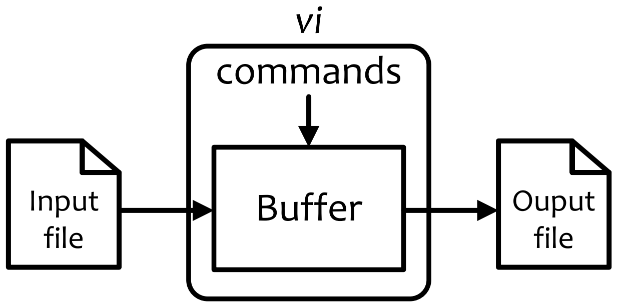


Figure 4.6: A simplified view of the *vi* program. *vi* reads in the input file and puts the content in the temporary buffer to manipulate it according to the commands from the user. *vi* commits to the output file from the buffer.

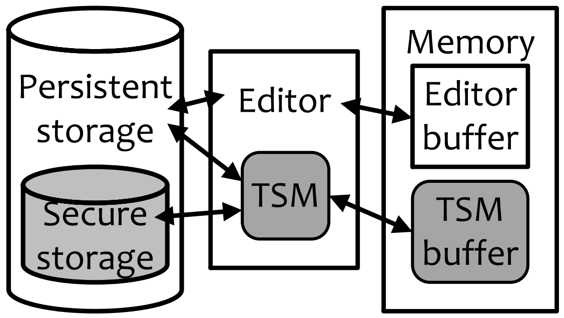


Figure 4.7: Partitioning an editor application into untrusted (white) and trusted (gray) parts. The TSM gets its own buffer to work with temporary decrypted data, and it can access both the secure storage where the policies are stored and normal storage where the protected (encrypted) content is stored.

As mentioned before, we dedicate a special TSM buffer for use only by the TSM to store and manipulate any temporary plaintext data it uses. All the data in the TSM buffer are tagged as secure data in the processor's on-chip caches. When secure data cache lines are evicted from on-chip caches out to the main memory, the SP hardware will ensure that they are encrypted and hashed, by a key that is derived from the DRK. The TSM buffer is used by the TSM to hold temporary decrypted lines of the protected content. In other words, the protected content remains encrypted inside all internal buffers of temporary files used by the editor, only decrypted by the TSM in the TSM buffer when the TSM is active. Figure 4.7 shows the interaction between the editor application (the trusted TSM and untrusted parts), the temporary buffers (SP-protected and unprotected), and the persistent storage (secure and normal).

To partition an application, we need to identify the entry points in and out of the TSM. We first categorize the commands available in *vi*. Figure 4.8 shows the flow chart we used to categorize the commands of *vi* into 5 groups. In this categorization process, we wish to identify whether or not a command is related to the opening/closing and

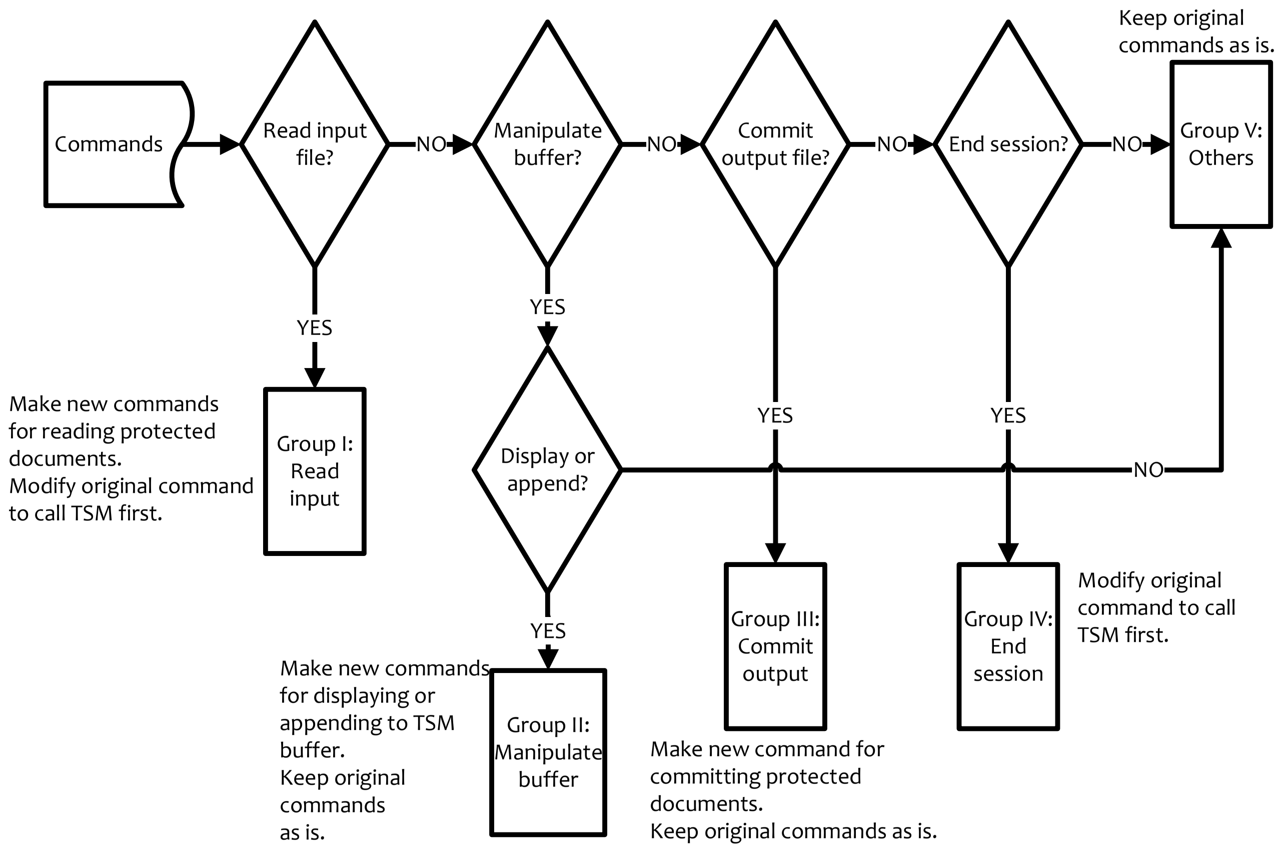

Keep original commands as is.
Commands
Read input file?
NO
Manipulate buffer?
NO
Commit output file?
NO
End session?
NO
Group V: Others
YES
YES
YES
YES
Make new commands for reading protected documents. Modify original command to call TSM first.
Group I: Read input
Display or append?
NO
Modify original command to call TSM first.
Group III: Commit output
Group IV: End session
YES
Make new commands for displaying or appending to TSM buffer. Keep original commands as is.
Group II: Manipulate buffer
Make new command for committing protected documents. Keep original commands as is.

Figure 4.8: Categorization of functions within an application for TSM protection.

Table 4.1: The groups of *vi* commands after categorization. The commands in **bold** are modified *vi* commands and the commands in *italics* are new commands. Commands in normal font are not changed.

Group I **Read input**	**Group II** **Manipulate buffer**	**Group III** **Commit output**	**Group IV** **End session**	**Group V** **Others**
ex	print	write	**quit**	abbreviate
tsm_ ex	read	*tsm_ write*		args
	tsm_ print			cd
	tsm_ read			delete
				⋮
				tsm_ create

reading/writing of a file, and whether or not a command is manipulating the data stored in *vi*'s buffer. These commands need to be identified since the TSM must be invoked to access the protected data when these commands are issued on a protected file. We modify the original command when a command changes the state of the TSM-protected data, e.g., the "quit" command terminates the current editing session and the TSM needs to be invoked to erase the plaintext in the TSM buffer. For commands that manipulate the editor buffer, we introduce new TSM commands that perform the same operation on the protected data in the TSM buffer, since the original command does not have access to the data in the TSM buffer. These are for the commands "print" and "read". Likewise, we introduce new commands that read a protected file ("ex") and commit the protected data back to the permanent storage ("write"). All other commands that are not relevant to the above (Groups I-IV) or commands that are not allowed to be performed on protected data, e.g., delete a string, are not modified (Group V). One special case is the *tsm_ create* command which turns existing documents into a protected document. This command is introduced to facilitate the data owner for creating protected documents.

Table 4.1 shows the commands in each group. In particular, we are interested in the commands that are relevant to our protected data, e.g., displaying the content of a file or appending new data to the original file, etc. The commands in **bold** (i.e., **ex** and **quit**) are modified *vi* commands and the commands in *italic* are new commands. These commands are the entry points to the TSM and are the only commands that can legitimately manipulate the plaintext within the TSM buffer. They start by bringing the processor into CEM and finish by exiting CEM, hence each of these commands is protected by the SP hardware to ensure they perform the desired functions. All other commands of *vi* remain unchanged. There are a total of 70 commands in the original *vi*, with 2 modified, 5 new ones added and the remaining 68 unmodified. The new and modified commands are described in more detail in Table 4.2.

Table 4.2: New and modified vi commands.

tsm_ex filename	Open a protected document.
tsm_print line_number	Display the contents of a protected document.
tsm_read filename	Append the contents of filename to current protected document.
tsm_write	Automatically re-encrypt the protected document (with any appended data) and update the length and the hash stored in the policy package.
tsm_create filename	Turn an existing document into a protected document.
quit & **ex**	End the current editing session. Erase the plaintext in TSM buffer, if any.

The above partitioning steps, although applied to *vi* specifically, can also be applied to other applications, with the goal of identifying the entry points of the TSM. We propose the following methodology for trust-partitioning an existing application:

1. Identify the security-critical information that needs to be protected.
2. Identify whether the information is transient data or persistent data. Transient data refers to the data that only exists during execution, such as a temporary buffer or program states, whereas persistent data are data stored in persistent storage, such as the disk.
3. Identify the input and output paths leading to and leaving from the protected information. These are the code paths that take certain input values and end up accessing the security-critical information, and the code paths that access the information and produce output values that are related to the information.
4. Relocate the protected information to the TSM buffer for transient data or the secure storage for persistent data. To relocate the information, one needs to modify or replicate existing code into TSM code to enter CEM and access the transient information through secure load/store instructions. For security-critical persistent data, they need to be turned into a protected data, e.g., using the *tsm_create* command, and be accessed through the TSM.
5. Modify the input and output paths using the new TSM functionalities. Since the security-critical part of the application has been turned into a TSM, the rest of the application needs to be modified to call the TSM at appropriate points within the application.

4.5 Security Analysis

We analyze the security of our proposed solution according to three main security concerns: confidentiality, integrity and availability.

4.5.1 Confidentiality

In the ORCON policy, the data owner is most concerned with the confidentiality of the sensitive content in the protected document – only the authorized recipients can have access to the decrypted content.

We first consider the case where the adversary is outside the trust group, e.g., the adversary does not have a legitimate device enabled with our architecture. The adversary can try to attack the system by intercepting the communication (1) when the data owner is sending the encrypted document over to the recipients, or (2) when the data owner's device is sending the policy package to the recipients' devices. The adversary does not gain any information in the first case since the document sent over the communication is encrypted, and we assume the use of strong cryptography. Similarly, the communication channel intercepted in the second attack is also encrypted, using the group encryption key, which is known only by a legitimate device within the same group.

The attacker can also steal one of the recipients' devices and try to impersonate the authorized recipient. In this attack, in order for the adversary to successfully authenticate himself as the authorized recipient, he must know, or have access to, the private key of the authorized recipient.

In the extreme case where the adversary is in the authorized recipient list – an *insider* attack – the adversary can access the contents of the document but has no way of digitally copying the contents to another file, since the plaintext document is only present in the TSM buffer during Concealed Execution Mode (CEM) and there is no command that allows direct memory copy of the plaintext from the TSM buffer to unprotected memory.

Although covert channels and side-channels are not included in the threat model in this thesis (Chapter 3), we briefly discuss it with regard to information leakage. We consider how side-channel and covert-channel would affect the security of our solution and how such issues should be mitigated in the implementation. First of all, the encryption algorithm employed in all encrypted data, including all of the input/output paths from/to the TSM (disk, keyboard, display, network, etc.) must have *semantic security* [48][3]. For example, the length of the ciphertext should be made unrelated to the length of the plaintext, e.g., by always padding zeroes to a fixed length, to prevent the attackers from knowing the size of the protected data. Furthermore, the encryption should employ chaining of the plaintext to prevent dictionary attacks on the ciphertext. Chaining of the plaintext blocks makes each ciphertext block depend on all previous plaintext blocks. Otherwise, identical plaintext blocks are encrypted into identical ciphertext blocks and attackers are able to learn the data patterns.

Even with semantic security measures, the ciphertext can still offer both side and covert channels that leak information, including timing and addressing channels. For example, even if the keyboard data is encrypted, the timing information can leak both the identity of the user, and likely the plaintext sequences. The addressing information can lead to traffic flow analysis for the network and the disk. In general,

[3]A system is said to be semantically secure if an adversary who knows the encryption algorithm and the ciphertext is unable to determine any information about the plaintext.

timing information can be blinded by fixed timing, albeit with performance penalty. However, the addressing information, e.g., Internet IP addresses, is difficult to block and out-of-scope for this thesis. Several layers of security measures would need to be employed on top of our solution to fully mitigate these side and covert channel attacks.

4.5.2 Integrity and Availability

The integrity of the protected document and the corresponding policy is enforced by the hash that ties together all the pertinent information of a policy-protected document (See Figure 4.3). The hash is stored in the secure storage, which is itself encrypted and integrity protected by the TSM using the keys accessible only to the TSM. The root of trust of the integrity of the secure storage is stored on the processor chip (SRH). Therefore, there is an integrity trust-chain from the protected content and policy package to the SRH, which does not depend on the potentially compromised OS.

This thesis does not directly address denial-of-service attacks, therefore if the adversary modifies or completely deletes the document, or the policy package in the secure storage, any access to the protected information is lost. Although it is easy to achieve such denial-of-service attacks, they are not considered detrimental since no security-critical information is leaked by these attacks. In fact, these attacks show the fail-safe nature of the access control implementation. Nevertheless, our architecture does provide intrinsic support for availability, in terms of the resiliency of the TSM to unrelated attacks to other software entities within the system. Since the trust chain consists only of the SP hardware and the TSM, attacks on the untrusted part of the application and/or the operating system do not prevent the TSM from enforcing its data protection functions.

4.6 Summary

In this chapter, we showed how to build applications to express and enforce an originator-controlled (ORCON) distributed information sharing policy for data, which is difficult to achieve with traditional MAC or DAC mechanisms. We leveraged the existing SP architecture to provide the runtime execution protection of the security-critical part of an application, i.e., the Trusted Software Module (TSM), and we design the generic TSM structure that can be extended to support different policies. We also demonstrated a proof-of-concept implementation using the *vi* application, having only to change a small fraction of its code base for the TSM. We also developed a methodology for *trust-partitioning* an existing application, which is useful generally for separating out the security-critical parts of an application.

The policy enforcement module in the TSM can implement any desired data confidentiality policy within a domain, so only one TSM for an application is needed to access any protected data object within a domain. Nevertheless, the approach taken

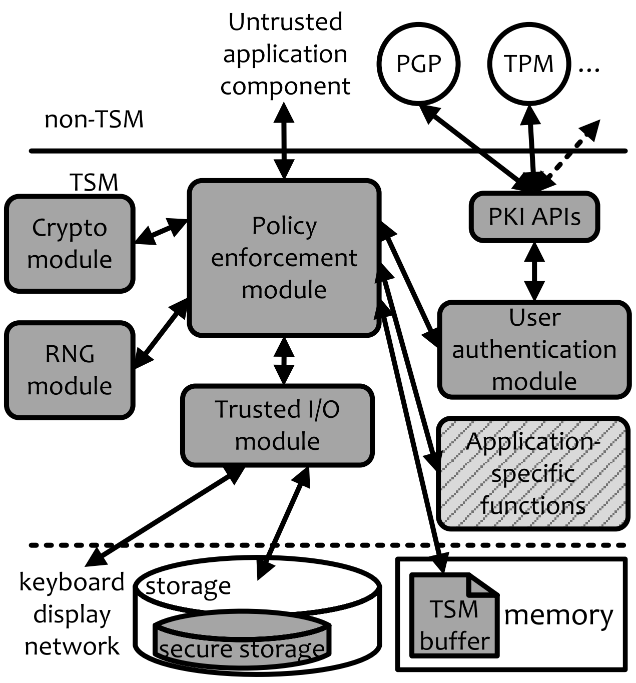


Figure 4.9: The TSM architecture including application-specific functionalities (shown in lighter gray with stripes).

in this chapter of protecting data by trusting a piece of application software, the TSM, that is protected by the hardware, has certain limitations:

1. The trust-partitioning methodology can only be applied *one application at a time*. In other words, every application that the authorized recipient would like to use to access the protected data would need to be partitioned before it can be used to access the data.
2. Although in our *vi* example, only around 10% of the total commands are either converted or introduced into the TSM, in practice, the more functionality one would like to operate on the protected data, the larger the portion of the application that would have to be included in the TSM, as shown by the "application-specific functions" block in Figure 4.9. For example, a modified *search* command would have to be provided by the TSM if the data owner allows the recipients to search within the protected document.

In order to address these limitations, this thesis explores solutions that are *application-independent*, which lead us to the next chapter on Policy-Protected Data *without* Trusted Application Software.

Chapter 5

Policy-Protected Data *without* Trusted Application Software

In the previous chapter, we explored the potential of providing data-centric protection with a trusted application software component. However, the inconvenience of having to partition every piece of application into trusted and untrusted parts, and the possibility of bloating the size of the trusted application software may hinder the adoption of such a method. In this chapter, we investigate a technique to protect data confidentiality in the absence of a piece of trusted application software. We propose DataSafe [30], a hardware-software architecture that provides data-centric protections independent of the applications that are accessing the data. Specifically, we make the following contributions:

- A new software-hardware architecture, DataSafe, to realize the concept of Self-Protecting Data. This architecture allows unvetted application programs to use sensitive data while enforcing the data's associated confidentiality policy. In particular, DataSafe prevents secondary dissemination by authorized recipients of sensitive data, protects data derived from sensitive data, and protects sensitive data at-rest, in-transit and during-execution.
- DataSafe architecture is the first to bridge the semantic gap between high-level policies and low-level hardware enforcements by automatically translating high-level policies expressed in software into hardware tags at runtime, without requiring modification of the application program.
- DataSafe provides efficient, fine-grained runtime hardware enforcement of confidentiality policies, performing derivative data tracking and nonbypassable output control for sensitive data, using enhanced dynamic information flow tracking mechanisms.

This chapter contains content from [30] as well as additional material.

5.1 Overview

This chapter proposes a new software-hardware architecture called DataSafe for protecting the confidentiality of data when processed by unvetted applications, e.g.,

Table 5.1: The problem space of secondary dissemination.

	Inadvertent	Malicious	
User	✓	✓	
Benign applications	✓	Not applicable	
		Explicit	Implicit
Malware	✓	✓	partial

programs of unknown provenance. It is based on the following key insights. First, the data owner (not the application writer) is the one most motivated to protect the data, and hence will be motivated to make some changes. Hence, in our proposed solution, the data owner must identify the data to be protected and must specify the data protection policy. The application program is unchanged and continues to deal with data only, and is unaware of any policies associated with the data. This gives the added advantage of our solution working with legacy code. The behavior of the application program must be monitored, to track the protected data as the application executes, and to ensure that the data's protection policy is enforced at all times. Table 5.1 summarizes the various types of secondary dissemination supported by DataSafe.

Second, we observe that while an authorized user is allowed to access the data in the context of the application and the current machine (or virtual machine), data confidentiality (beyond this session) is protected as long as any output from the current machine is controlled according to the data's protection policy. Output includes the display, printing, storing to a disk, sending email or sending to the network. Furthermore, any data derived from sensitive data must also be protected. Hence, our DataSafe solution proposes continuous tracking and propagation of tags to identify sensitive data and enforce nonbypassable output control.

DataSafe architecture realizes the concept of *self-protecting data*, data that is protected by its own associated policy, no matter which program, trusted or untrusted, uses that data, unlike in the previous chapter where the protected data has to be accessed through a Trusted Software Module. The data must be protected throughout its lifetime, including when it is at-rest (i.e., in storage), in-transit, and during execution. The data protection must apply across machines in a distributed computing environment, when used with legacy applications or new unvetted programs, across applications and across the user and operating system transitions. A self-protecting data architecture must ensure that: (1) only authorized users and programs get access to this data (which we call *primary authorization*), (2) authorized users are not allowed to send this data to unauthorized recipients (which we call *secondary dissemination by authorized recipients*), (3) data derived from sensitive data is also controlled by the data's confidentiality policy, and (4) confidentiality policies are enforced throughout the lifetime of the data.

We assume that the first problem of primary authorization can be solved by well-known access control and cryptographic techniques, and will not discuss this further in this chapter. Rather, this chapter tackles problems (2), (3) and (4). Problem (2), the secondary dissemination by authorized recipients, is especially difficult and dangerous, since an authorized recipient of protected information (passing the primary authorization checks) can essentially do anything he/she wants with it in commodity systems today.

Secondary dissemination of protected information can be by an authorized user or by an application, and can be either malicious or inadvertent. A malicious user example could be a confidentiality breach by an insider, such as a nurse in a hospital trying to sell the personal information of some celebrity admitted to the hospital whose medical records he or she is authorized to access. An example of inadvertent secondary dissemination of confidential data could be a doctor trying to send his/her family a vacation plan as an attachment, but accidentally attaching some patient's psychiatry record instead. When programs are the culprits rather than users, a malicious, privacy-stealing malware, installed on an authorized user's machine through social-engineering, could send out sensitive information, or a benign application may contain bugs that could be exploited to leak sensitive information. In DataSafe, we enforce nonbypassable output control to prevent such breaches by authorized users.

Data derived from sensitive data must also be tracked and protected. An unvetted application program can be *designed* to leak sensitive information. It could transform or obfuscate the sensitive data. For example, a credit card number could be transformed and obfuscated into several paragraphs of text, before being output from the application, so that no sequence of numbers resembling a credit card number can be identified. This requires somehow tracking the information flows from protected data to other variables, registers or memory locations, across applications and system calls, and across combinations of data such as in mashups. In DataSafe, we argue that such continuous tracking of sensitive data, through any number of transformations, requires some form of dynamic information flow tracking.

For confidentiality policies to be enforced throughout the lifetime of the protected data, DataSafe uses encrypted packages to transmit data between DataSafe and non-DataSafe machines in a distributed environment, as illustrated by Figure 5.1. A data owner wants the sensitive data to be accessed and used by authorized users according to the data's associated security policy. However, authorized users or applications can maliciously or inadvertently compromise the confidentiality of the protected data by distributing (or leaking) the plaintext of the sensitive data to unauthorized users. DataSafe addresses this problem by: (1) controlling the use of data and preventing leakage on a DataSafe machine while data is used by authorized users (Case A), (2) ensuring secure data transfer to both DataSafe and non-DataSafe machines, and in particular that no protected data is ever sent in plaintext outside the machine (Case B), (3) enabling only authorized users to use protected data on DataSafe machines (Case C, D), and (4) preventing any user from accessing protected data (in plaintext) on a non-DataSafe machine (Case E, F). This last case is restrictive, in terms of availability, but provides fail-safe confidentiality protection within the current ecosystem.

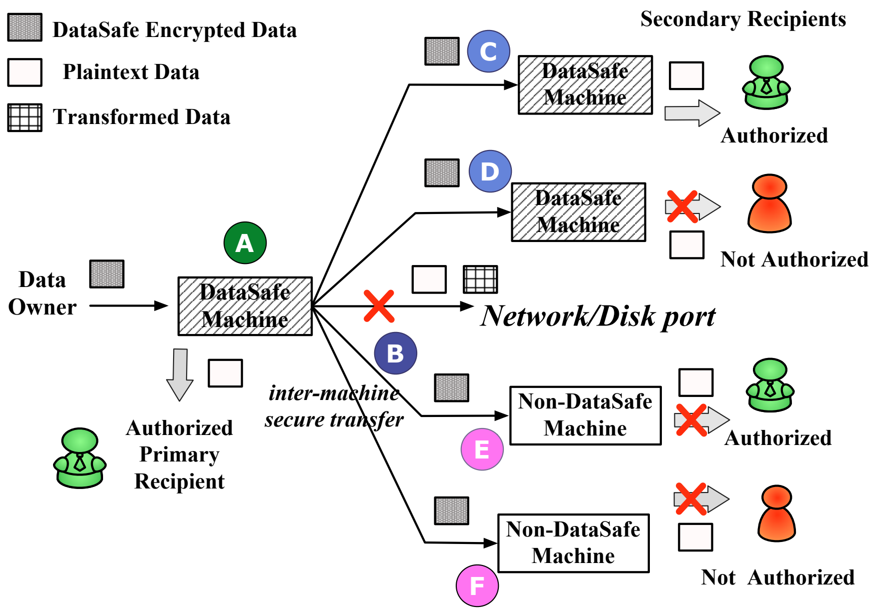


Figure 5.1: DataSafe architecture protects data confidentiality across machines (new and legacy) and users (authorized and not authorized).

(With built-in processor security, the idea is that eventually, all future ubiquitous computers will include DataSafe features.)

Figure 5.2 illustrates the key ideas on how DataSafe enables *self-protecting data*. To protect data-at-rest and data-in-transit, DataSafe uses strong encryption to protect the data, while ensuring that only legitimate users get access to the decryption key. For data-during-execution, DataSafe creates a *Secure Data Compartment (SDC)* where untrusted applications can access the data, as they normally would. When data (e.g., a protected file) is first accessed by an application, DataSafe software (Policy/Domain Handler) does a primary authorization check, before translating the data's high-level policy to concise hardware "activity-restricting" tags. The DataSafe hypervisor then creates Secure Data Compartments (SDC), within which sensitive data is decrypted for active use by the application. Each word of the protected data in the SDC is tagged with a hardware activity-restricting tag. From then on, DataSafe hardware automatically tracks the data that initially comes from SDCs, propagating the hardware tags on every processor instruction and memory access. By restricting output activities based on the hardware tags, DataSafe prevents illegitimate secondary dissemination of protected data by authorized recipients, even when the data has been transformed or obfuscated. The hardware tag propagation and output control is done without the knowledge of the applications software, and applies across applications and across application and operating system transitions. We prototype our software-hardware architecture and show that it indeed prevents confidentiality breaches, enforcing the data's confidentiality policy, without requiring any modifications to the third-party applications.

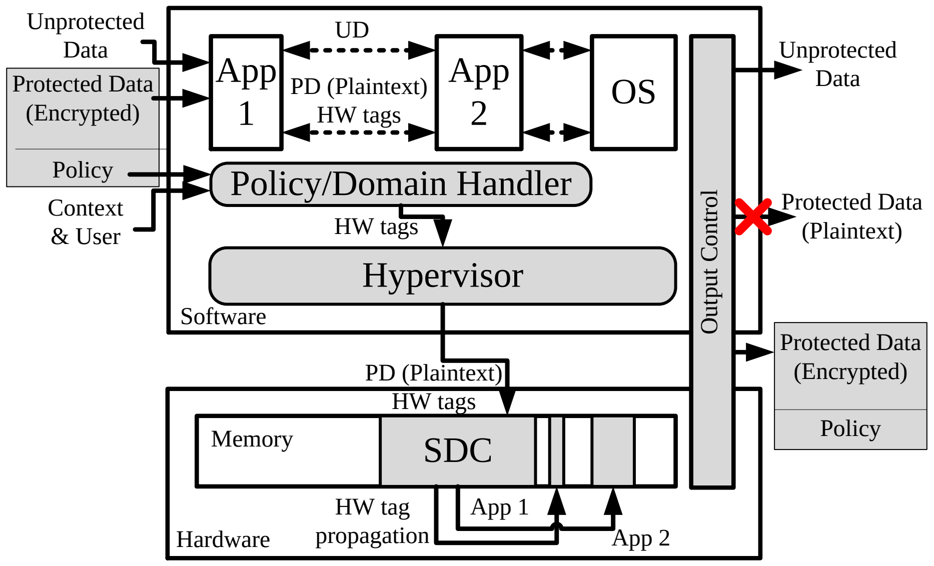


Figure 5.2: Software-hardware monitoring of Protected Data (PD) in DataSafe architecture. Unprotected Data (UD) is unchanged. Since the hardware tags of the Protected Data are tracked and propagated at the physical memory level by the hardware, this allows seamless tracking across applications and across application-OS boundaries, as illustrated by the top row of boxes. (Gray indicates DataSafe additions).

5.2 DataSafe Architecture

We first describe the overall operation of enforcing data confidentiality and next describe how the DataSafe software components achieve the automatic translation of high-level security policies without having to modify the third-party applications. Lastly we show how the DataSafe hardware components achieve continuous runtime data tracking with output control.

5.2.1 Overview

DataSafe architecture consists of software and hardware components, as shown in Figure 5.3. The DataSafe software has the following responsibilities: (1) to translate protected data's high-level security policy into hardware enforceable tags, (2) to create a secure data compartment (SDC) by associating the tags with the plaintext data in the memory, and (3) to achieve application independence by enabling third party applications to use the data without having to modify them.

The key challenge in the tag generation process is that the hardware tags must accurately reflect the permissions and prohibitions required by the data's security policy. Tags for a given policy are not fixed, but rather they change depending on the context within which the policy is interpreted. In DataSafe software, a policy/domain handler is responsible for translating policies to tags, and the hypervisor is responsible for associating hardware tags with data to create an SDC.

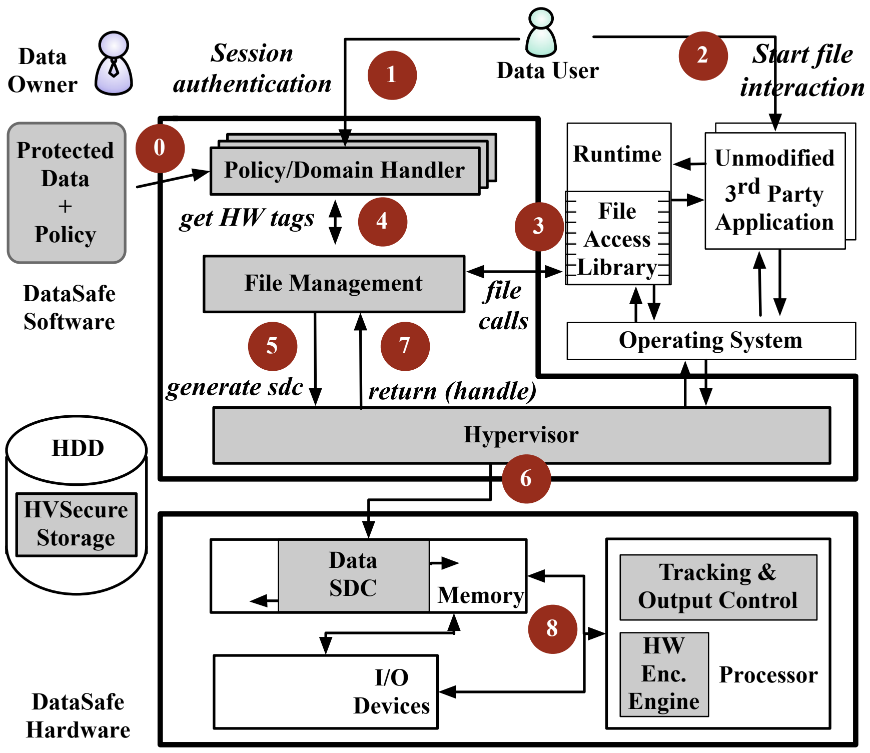


Figure 5.3: The software and hardware components of DataSafe. The gray parts are new and trusted DataSafe components, while the striped file access library is modified but untrusted. All other software entities including the unmodified third-party applications and the operating system are assumed to be untrusted.

Both the hypervisor and the policy/domain handlers are assumed to be trusted code. The hypervisor maintains its own secure storage (protected by hardware) to store keys and other data structures. The hypervisor is protected by the most-privileged level in the processor and directly protected by the hardware (e.g., as in Bastion [25]). The policy/domain handler is run in a trusted virtual machine protected by the trusted hypervisor. Alternatively, for a simple, fixed policy/domain handler, it can be incorporated as part of the trusted hypervisor, to reduce the total number of software entities to be protected. We describe the protection of the trusted hypervisor in Section 5.2.4.

DataSafe Operation

DataSafe operates in four stages – Data Initialization, Setup, Use, Cleanup and Write-back, as explained below.

Data Initialization. During the Data Initialization stage, represented by Step 0 in Figure 5.3, a DataSafe package containing the (encrypted) data to be protected, along with its associated policy, is brought into a DataSafe enabled machine. The details of creation and unpacking of DataSafe packages are explained in Section 5.2.8.

Setup. In the Setup stage, a secure data compartment (SDC) is dynamically created for the data file. An SDC consists of hardware enforceable tags defined over a memory region that contains decrypted data. Hardware tags are generated from the policy associated with the data. Once an SDC is created for a file, users can subsequently use the data file via potentially untrusted applications, while the hardware ensures that the data is used in accordance with the associated policy.

The Setup stage takes place during Steps 1-6, as shown in Figure 5.3. In Step 1, a user starts a new session by providing his/her credentials, and is authenticated by the policy/domain handler. The user authentication information and other system and environment properties constitute the context that is collected by the policy/domain handler. During the session, the user requests file interaction using a third-party application, as shown in Step 2. The third-party application's request is forwarded to the file management module in Step 3 by the modified file access library of the runtime. In step 4, the file management module requests the policy/domain handler to provide the hardware tags to be set for the file. The policy/domain handler validates the policy associated with the data file taking into consideration the current context (i.e., the user/session properties, data properties and system/environment properties), and generates appropriate hardware tags for the data file.

In Step 5, the file management module requests the hypervisor to create an SDC for the data file with the corresponding hardware tags. In Step 6, the hypervisor decrypts the data file, and creates an SDC for the data file associating the appropriate tags with each word in the SDC. In Step 7, the file handle of the SDC is returned back to the policy/domain handler and the execution is returned back to the application.

Use. In the Use stage, the DataSafe hardware tags each word of the protected data in each SDC and persistently tracks and propagates these tags, as shown by Step 8. Once an SDC is set up for a data file, in accordance with the session properties, any third-party application can operate on the protected data as it would on any regular machine. The DataSafe hardware will ensure that only those actions that are in conformance with the data-specific policy are allowed.

Cleanup and Writeback. After the application finishes processing the data, the DataSafe hypervisor re-packages the protected data and the policy if the data was modified or appended to, re-encrypts the protected data, removes the associated tags within the SDC, then deletes the SDC.

5.2.2 Runtime Translation of Expressive Software Policy to Hardware Tags

The two DataSafe software components, the policy/domain handlers and the hypervisor, take a high-level policy specified in a policy model, translate the policy into the hardware enforceable tags and create an SDC for the protected data, as shown in Figure 5.4.

DataSafe employs a two step process for the hardware tag generation: (1) a security policy is interpreted to determine what high-level actions are permitted (policy interpreter), and (2) depending on the high-level actions permitted, the appropriate

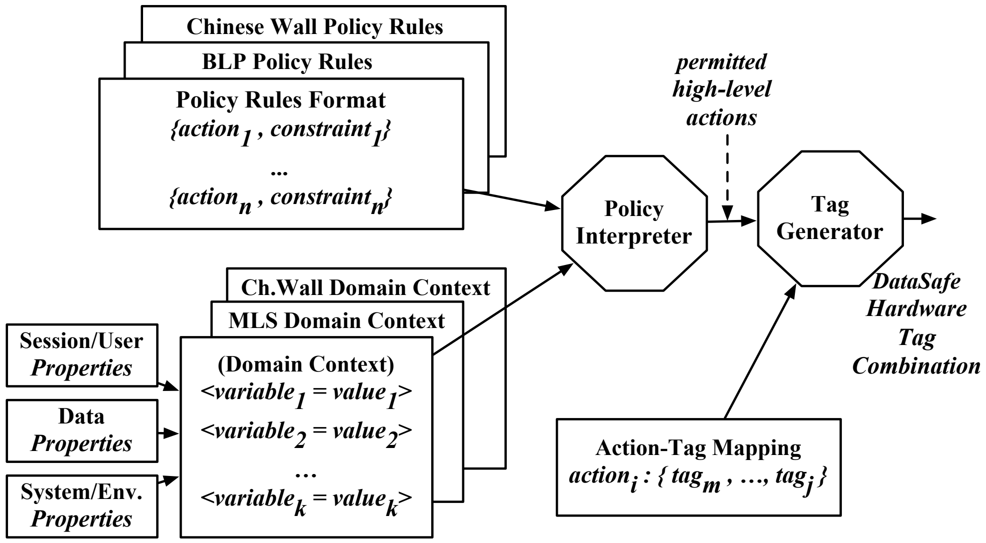

Chinese Wall Policy Rules
BLP Policy Rules
Policy Rules Format
{action$_1$, constraint$_1$}
...
{action$_n$, constraint$_n$}
permitted high-level actions
Policy Interpreter
Tag Generator
DataSafe Hardware Tag Combination
Ch.Wall Domain Context
MLS Domain Context
Session/User Properties
Data Properties
System/Env. Properties
(Domain Context)
<variable$_1$ = value$_1$>
<variable$_2$ = value$_2$>
...
<variable$_k$ = value$_k$>
Action-Tag Mapping
action$_i$: { tag$_m$, ..., tag$_j$ }

Figure 5.4: Translation from high-level policies to hardware tags.

hardware tags are chosen (tag generator). DataSafe is designed to be generic, supporting multiple policy languages and policies such as Bell-LaPadula (BLP), Biba, etc (see Page 49 for their definitions). However, we will not go into the details of the various policy models and their implementations in this thesis.

A policy is expressed and interpreted in terms of a context, which typically includes information about user properties, data properties and system properties necessary to interpret the policy appropriately, in a given domain. For example, a BLP policy will require the user's security clearance and the data's security classification, whereas a Role-Based Access Control (RBAC) policy will require the user's role. The context information is collected and stored in the form of $\{variable, value\}$ pair. The policy and the context information are then fed to the policy interpreter, which determines what high-level actions are permitted by the policy, on the data, under the current context values. If no high-level action is permitted, then access is denied at this point. If this set is non-empty, it means that the user has authorized access to the data, but is expected to use the data only in the manner defined by the permitted action set. The permitted action set is then used to calculate the hardware tags, and to generate the SDC for the data.

Policy Model. Our policy model consists of a set of restricted actions $\{ra_1, ra_2, ra_3, ..., ra_n\}$, where each restricted action includes an action associated with a constraint, represented by $ra_i = \{action_i, constraint_i\}$. The *action*, is a high-level action such as "`read`", "`play`", "`view`", etc. The *constraint*, is a predicate defined in terms of context variables. A context is defined by a set of variables $\{v_1, v_2, ..., v_n\}$, that represents user, data and system properties. A given constraint evaluates to either *true* or $false$ based on the values of the context variables. For a given restricted action, $ra_i = \{action_i, constraint_i\}$, if $constraint_i$ evaluates to *true*, then $action_i$ is permitted, otherwise it is not permitted. Algorithm 1 describes the algorithm for calculating the set of restricted actions for a set of DataSafe-protected

Algorithm 1: Calculate the restricted actions for a protected data element.

```
Input: $P$: Set of protected data; $RA$: Set of restricted actions
Output: $ra[p]$ : Set of restricted actions for each protected data $p \in P$
foreach $p \in P$ do
    Set the context $v$ with $p$;
    foreach $action \in RA$ do
        Query the policy: Is $action$ allowed under context $v$?;
        if $no$ then
            $ra[p] \leftarrow ra[p] \cup action$;
        end
    end
end
```

data. The context variables can be a arbitrary set of values required by a specific policy. For example, the name of the caller function, the current date and time or the current user name could be used as context variables. These context values can be retrieved from the policy/domain handler itself, from the operating system through system calls or from the hypervisor through hypercalls.

Permitted Policy-Level Actions to Hardware tags. For every policy, the semantics of its high-level actions, described within the policy, have a specific interpretation in terms of hardware-level actions. Based on this interpretation, every high-level action maps to a set of hardware tags. At present, the DataSafe prototype supports six hardware tag values, as shown in Column 4 of Table 5.2, but the architecture can support more tag values. Hardware restriction tags are expressed in the form of a bit vector, where each bit, when set to 1, corresponds to a type of restriction. The hardware tags are restrictive, which means that if a particular tag bit is set, that particular hardware-level action is prohibited. For example, if the tag bit $0x01$ is set for an SDC, the DataSafe Hardware will prevent any application and the OS from copying that SDC's data to the display output. On the other hand, if the policy permits the action "`view`", then tag $0x01$ should *not* be set. Hence, for a given policy interpretation, the set of tags corresponding to the permitted actions are *not* set, and the rest of the tags are. The tag generation process is independent of whether a policy is attached to a particular datum, or it applies system wide to all data items. Hence, DataSafe can support both mandatory and discretionary access control policies.

DataSafe hardware tags are divided into three categories: (1) Access, (2) Transient Output, and (3) Persistent Output tags. Tags in the Access category, which include *write* (or edit), *append* and *read*, prevent in-line modification, appending or reading of an SDC, respectively. The tags in the Transient Output category refer to the output devices where the lifetime of the data ends after the data is consumed by the device, e.g., the display or the speaker. The Persistent Output category deals with the output devices where data remain live after being copied to those devices, e.g., network or disk drives. If an application or a user, after gaining authorized access to protected

Table 5.2: The correspondence between policy-prohibited activities and the hardware tags that restrict that activity.

Actions	**Category**	**Restriction**	**Tag**
Edit	Access	No write to SDC	0x08
Append	Access	No append to SDC	0x10
Read	Access	No read from SDC	0x20
View	Transient output	No copy to display	0x01
SendPlaintext	Persistent output	No copy to network	0x02
SavePlaintext	Persistent output	No copy to disk	0x04

Algorithm 2: Calculate the tag combination for a given set of restricted actions.

```
Input: $ra[p]$: Set of restricted actions for protected data $p$; $T$: Set of tags; $F$:
       mapping of restricted actions to tag values
Output: $t_{ra[p]}$: Tag combination representing the set of restricted actions $ra[p]$
        for protected data $p$
$T_{include} = \phi$;
foreach $action \in ra[p]$ do
    $T_{include} \leftarrow T_{include} \cup F(action)$;
end
$t_{ra[p]} \leftarrow$ 0x0;
foreach $tags\ t \in T_{include}$ do
    $t_{ra[p]} \leftarrow t_{ra[p]} + t$;
end
```

plaintext data, saves the data in plaintext form on a disk drive, or sends the plaintext over the network, the data's confidentiality is permanently lost. Most policies don't explicitly mention the requirement to prevent such activities, but rather assume that the authorized user is trusted not to illegally leak the data out. In order to enforce this critical and implicit assumption, in DataSafe systems, these two tags, No copy to network and No copy to disk, are always set for all *confidentiality-protected* data for all policies, except for policies that have explicit declassification rules. Algorithm 2 shows the algorithm to calculate the set of hardware tag values for a set of restricted actions for DataSafe-protected data. The DataSafe prototype uses a mapping function F as described by Table 5.2 to map the restricted actions to the hardware tag values.

Policy Examples

To get a concrete idea of how the policy translation works, we consider two examples: (1) a simple data-protection policy for protecting encryption keys such as AES keys or private keys such as RSA keys (e.g., against the memory search attacks to find keys, as described in Section 1.1), and (2) a Bell-LaPadula (BLP) policy defined over a multi-level security (MLS) environment.

Table 5.3: A simple key-protection policy.

Context Variables:
caller $\in$ {`libcrypto`, all others}

Action to Tags Map:
read $\Rightarrow$ {No read from SDC}
write $\Rightarrow$ {No write to SDC, No append to SDC}
leak data $\Rightarrow$ {No copy to display, No copy to disk, No copy to network}

Key-Protection Policy:
$ra_1 := \{action := read,\ \ constraint := (caller ==$ `libcrypto`$)\}$,

Use Case:
Actions permitted: $\{read\}$
Actions prohibited: $\{write, leak\ data\}$
Tags set: {No write to SDC, No append to SDC, No copy to display, No copy to disk, No copy to network}

For a key-protection system, it is desirable that the keys can be used but never seen, either by explicitly displaying the key value, by saving the plaintext key value on the disk or by sending the plaintext key value out on the network to another machine. Therefore, a simple key-protection policy may state: *"An authorized user can only use a key value for encryption/decryption purposes and that the plaintext key value is not allowed to be displayed on the screen, saved on the disk, sent to the network or changed to another value."* The representation of this policy in our standard policy model is shown in Table 5.3. For the DataSafe system to recognize that the request is indeed from an encryption/decryption process, we install a trusted cryptographic library (`libcrypto` in the example in Table 5.3) in the system. When the library is invoked by an application to perform encryption/decryption, it notifies the domain/policy handler that the application is requesting to access the key for encryption/decryption purposes. The domain/policy handler can then query the policy based on the restricted actions and verify the satisfaction of the constraint, before granting access to the key. Note that an application now has to call the trusted crypto library for the key access and will not be able to use the protected key if the application performs its own cryptographic operations. Table 5.3 also shows an example use case where only the *read* action is permitted to use a protected encryption key.

As a second example, we describe how the Bell-LaPadula (BLP) policy can be implemented for a Multi-Level Security (MLS) system. In a MLS system, each user has a *Security Clearance* and each data item has a *Security Classification.* Both properties range over the ordered set {Top Secret > Secret > Confidential > Unclassified}. The BLP policy states: *"A user at a security clearance x can only read data items with security classification y such that $y \leq x$ (no read up), and can write only to data items*

Table 5.4: The Bell-LaPadula (BLP) policy expressed in DataSafe.

Context Variables:
sec_clear $\in$ {Top Secret, Secret, Confidential, Unclassified}
sec_class $\in$ {Top Secret, Secret, Confidential, Unclassified}

Action to Tags Map: *read* $\Rightarrow$ {No copy to display, No read from SDC} *write* $\Rightarrow$ {No write to SDC, No append to SDC} *leak data* (*implicit*) $\Rightarrow$ {No copy to disk, No copy to network}
BLP Policy: $ra_1 := \{action := read,\ \ constraint := sec\_class \leq sec\_clear\}$, $ra_2 = \{action := write,\ \ constraint := sec\_class \geq sec\_clear\}$

Use Case 1: *sec_clear* := Secret, *sec_class* := Confidential
Actions permitted: {*read*}
Actions prohibited: {*write*, *leak data*}
Tags set: {No write to SDC, No append to SDC, No copy to disk, No copy to network}

Use Case 2: *sec_clear* := Secret, *sec_class* := Top Secret
Actions permitted: {*write*}
Actions prohibited: {*read*, *leak data*}
Tags set: {No read from SDC, No copy to display, No copy to disk, No copy to network}

with security classification z such that $z \geq x$ (no write down)".[1] The representation of this policy in our standard policy model is shown in Table 5.4.

The context variables *sec_clear* represents *Security Clearance* and *sec_class* represents *Security Classification*. BLP has *read* and *write* as high-level actions, while *leak data* is an implicit action. Each action corresponds to the hardware tags as shown. The BLP policy is the set of restricted actions $\{ra_1, ra_2\}$, where the constraints are expressed over context variables *sec_clear* and *sec_class*.

In Use Case 1 of Table 5.4, action *read* is permitted according to the BLP policy (no read up), since the subject's security clearance (Secret) is higher than the object's security classification (Confidential) and hence the *read* tag is not set, while *write* and *data leakage* tags are set, to prevent the object from being modified and leaked out of the system.

In Use Case 2, the subject is allowed to write to the object (no write down) but not allowed to read, and hence the *write* tag is not set, while *read* and *data leakage* tags are set to prevent the object from being read and leaked out of the system.

[1] A Biba policy states the dual of BLP as follows: *"A user at a security clearance x can only read data items with security classification y such that $y \geq x$ (no read down), and can write only to data items with security classification z such that $z \leq x$ (no write up)".*

Table 5.5: Example entries of the active SDC `sdc_list` software structure.

ID	Virtual address	Size	Tag
id1	vaddr1	size1	0x08
id2	vaddr2	size2	0x1C

5.2.3 Unmodified Applications

In DataSafe, the confidentiality-protection policy is defined for the data and packaged with the data (see Section 5.2.8), not defined by a particular application or its programmer. In other words, the data's policy is enforced no matter which application is accessing the data; therefore, applications are agnostic of DataSafe's operation and do not have to be modified to work with DataSafe. Only the file access library in the runtime or the interpreter has to be modified to redirect file calls of the application to the file management module of the DataSafe Software. Furthermore, DataSafe-protected data are protected with the SDCs, where the SDCs are defined at the hardware level, the layer below any software entity.

This is one of the key design features of DataSafe – defining the SDC over the physical machine memory, instead of the virtual memory. This enables us to achieve application independence and cross boundary data protection. Applications access their data through virtual memory. Once an SDC is created in the physical memory, an application can access the data within the SDC by mapping its virtual memory to the SDC in the physical memory. This data can be passed among multiple applications and OS components.

Once the hardware restriction tags are determined for a given data file, DataSafe associates those tags with the memory region allocated to the file, without having to change how the application accesses the protected data. Such an association is achieved by a secure data compartment (SDC). The DataSafe hypervisor is responsible for the creation, maintenance and deletion of SDCs, and maintains a software SDC list as shown in Table 5.5. An SDC is a logical construct defined over a memory region that needs to be protected, independent of the application. Every SDC has a start memory address, a size, and a tag combination specifying its activity-restricting rules with which the data within the SDC are protected.

For applications to access the DataSafe-protected data in an SDC, we modify the application file access library to redirect the access requests from the applications to the policy/domain handler(s), as shown previously in Figure 5.3. The modified file access library does not have to be trusted. In case the access request is not redirected by a malicious library for protected data, only encrypted data will be available to the application, which is a data availability issue instead of a confidentiality breach. We describe our modified file access library in more detail in Section 5.3.1.

SDCs can be defined at different granularities. DataSafe can define different types of SDCs over different parts of the data object. For example, different sections of a document, different tables in a database, or different parts of a medical record need different types of confidentiality protection. Correspondingly, the policy for the protected file must specify the offset of the different parts with their respective policies, such that the policy/domain handler can request the hypervisor for different SDCs for the different parts of a file.

5.2.4 Protecting the Hypervisor and Its Storage

In addition to assuming that the hypervisor is launched with boot-time integrity protection, the trusted hypervisor is also supplied with persistent secure storage, enabled by four registers inside the processor in Figure 5.5. This persistent secure storage is an encrypted and hashed area of non-volatile storage where the hypervisor can keep software "registers" and other security-critical information such as the public-private key pairs and certificates for the domain(s) it belongs to. These four registers, modeled after Bastion's hypervisor protection [25], ensure that no other software, including a malicious hypervisor, can access a given hypervisor's persistent secure storage.

The `hv_id` (hypervisor identity) register contains the hash of the hypervisor calculated at bootup time, and identifies the loaded hypervisor. The `hv_id` register is only written by the processor when invoked through dynamic secure bringup instructions, such as the `secure_launch` in the Bastion architecture [25], Intel `SENTER` instruction or the AMD `SKINIT` instruction used to bring up TrustVisor [66].

The hypervisor's persistent secure storage is cryptographically secured by three non-volatile registers: its confidentiality is protected by encryption with a key stored in the `pss_key` register, and its integrity by a hash tree with the root secured in the `pss_hash` (persistent secure storage hash) register. The identity of the hypervisor that created this persistent secure storage is kept in the `pss_owner` (persistent secure storage owner) register. This hypervisor storage is persistent in that it survives power on-off cycles.

Upon a subsequent bootup of a potentially malicious hypervisor, the identity of this hypervisor is calculated and stored in the `hv_id` register. In order to access any previously stored hypervisor storage, this new `hv_id` value must be equal to that in the `pss_owner` register. This ensures that the `pss_key` and `pss_hash` registers are only unlocked by the hardware when the loaded hypervisor is the rightful owner of the persistent secure storage. The hypervisor can then verify the integrity of the secure storage by verifying `pss_hash` and then use `pss_key` to decrypt its secure storage.

For protecting the dynamic execution of the trusted hypervisor, several existing memory integrity protection techniques exist [24, 42, 46, 79] and they can be applied together with DataSafe to ensure a secure runtime environment for the trusted hypervisor. Furthermore, the trusted hypervisor can be employed to protect the policy/domain handler, similar to a Bastion-protected software module [23, 25]. Note that the policy/domain handler is not application-specific like the Trusted Software Module presented in Chapter 4, where the policy is only enforced for a particular application accessing protected data, but rather the policy/domain handler enforces the policy regardless of which application is accessing DataSafe-protected data.

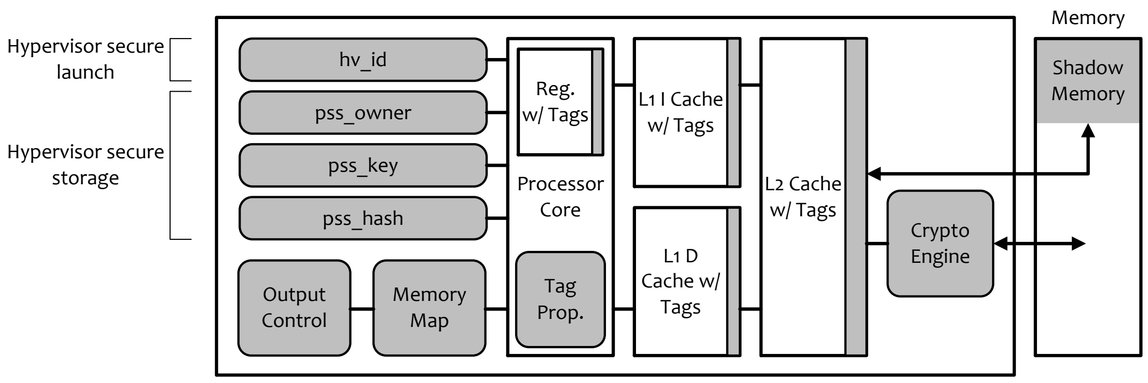


Figure 5.5: The DataSafe hardware components (gray).

5.2.5 Continuous Runtime Data Tracking

In order to provide continuous runtime protection for the protected data within an SDC while the application is executing, we use hardware mechanisms to track each word of the protected data throughout the execution of the untrusted application. DataSafe extends each 64-bit data word storage location with a k-bit SDC ID and a j-bit tag. The shadow memory shown in Figure 5.5 is a portion of the main memory set aside for storing the tags. It is a part of the hypervisor secure storage, which the DataSafe hardware protects and only allows the hypervisor to access. The hardware tag is set by the hypervisor when an SDC is requested to be set up by the policy handler. Note that only the hypervisor has read/write access to the shadow memory for adding and deleting the tags for the SDCs.

To track and monitor where the protected data resides in the system, we propagate the tags along with the data from within the SDC as it goes outside the SDC to other parts of memory. First, the hardware tag bits are propagated from the shadow memory to the last level on-chip cache, when a cache line is brought from the main memory due to a cache miss. The same tag bits are copied throughout the cache hierarchy, i.e., up to the level-1 data cache. The general purpose registers in the processor are also extended with the ability to propagate the tag bits. On memory load instructions, the tag bits are copied from the level-1 data cache to the destination register.

Each instruction executed in the processor performs tag propagation operations along with its arithmetic or other operations. This way the hardware restriction tags can track sensitive data even if the data has been transformed or encoded by the application. We use the principles of existing information flow tracking techniques [33], where the source tag bits are propagated to the destination register as long as the source register has a nonzero tag bit. In the case where both of the source registers have nonzero tag bits, we take the union of the two tag bits to give the destination register a more stringent policy[2]. For load instructions, the union of the tag of the source address register and the tag of the source memory data is propagated to the

[2]Special cases such as zeroing a register (e.g., "`xor %eax, %eax`" on x86) are treated differently. For example, the destination tag is cleared in this example.

tag of the destination register. For store instructions, the union of the tag of the source data register and the tag of the source address register is propagated to the tag of the destination memory address. Thus, the tag propagations for load and store instructions account for the index tag for table lookups. For integer arithmetic and multiply and divide instructions, the tag is a combination of the tag of the first source register, the tag of the second source register, the tag of the condition code register[3] and the tag of other registers if necessary, e.g., the y register[4] for the SPARC architecture. The tag of the condition code register is also updated if the instruction has these side-effects. The detailed descriptions for each instruction type and the propagation rules used by DataSafe are given later in Chapter 6 in Table 6.1.

If both of the source registers are tagged with the same SDC ID, the destination register is also tagged with this SDC ID. If they are not from the same SDC, we assign a reserved ID tag of $2^k - 1$. Since the resultant data does not belong to either of the two source SDCs, the SDC IDs are not combined; rather a special tag is substituted to indicate that this is an intermediate result. These intermediate results are cleared up with zeroes by the hypervisor when there is no active SDC in the system, e.g., when the user logs out of a session. For the tags of the data within an SDC, the hypervisor clears those tags when the SDC is deleted; whereas for the tagged data that propagates outside of SDC regions, the hypervisor zeroes those data locations in the background after the particular SDC is deleted.

The tag propagation rules described above handle explicit information flow from the data within an SDC, where the destination operands receive direct information from the source operands. There are also cases where the destination operand receives information from the source operand(s) through a third medium, e.g., the integer condition code or branch instructions. This kind of information flow is implicit but can be exploited to leak information. A vanilla dynamic information flow tracking system without considering such information flow would lead to false-negatives since information could be leaked without being tagged. However, a naive approach that tags any instruction that is dependent on the branch condition's tag may lead to an impractically large amount of false-positives [16, 55]. Such implicit information flows are discussed in more detail in Chapter 6.

5.2.6 Hardware Output Control

DataSafe hardware checks to see whether copying the data to another memory location or output device is allowed, or whether writing to memory locations within the SDC is allowed, according to the hardware tags. In particular, hardware checks if a memory location to be written to is a memory-mapped output device, and enforces output control according to the tag of the word being written.

We introduce a new hardware structure inside the processor: the output memory map, `mem_map`. The `mem_map` is only accessible to the trusted hypervisor. It stores

[3]To avoid being overly conservative, one could implement one tag for each bit in the condition code register, e.g., one tag for the zero flag and another tag for the overflow flag.

[4]The y register is used for storing the upper 32-bit result in multiplication and the remainder in division in SPARC architectures.

Table 5.6: Example entries of the output memory map `mem_map` hardware structure.

Start addr	End addr	Mask
`addr1`	`addr2`	(display) `0x01`
`addr3`	`addr4`	(network) `0x02`
`addr5`	`addr6`	(disk) `0x04`

memory-mapped I/O regions and I/O ports to enable the hardware to know if a memory store instruction is attempting to perform output. It is checked on the destination address of memory store instructions, or any other instructions that write to an output device (e.g., `in` and `out` instructions in x86 architecture), to see if there is a violation of the output policy specified in the tag associated with the data to be written.

Table 5.6 shows example entries in the `mem_map` hardware structure. The device mask is a bit mask which indicates its functionality e.g., display, speaker, USB storage, NIC, etc. Two devices having the same functionality would have the same mask value. In our DataSafe prototype, the mask is designed to match the activity-restricting bits in the hardware tags (Table 5.2). Therefore, the hardware output control unit performs an `xor` of the tag value from the data with the mask read out from the `mem_map`. A non-zero result indicates an output violation, whereas a zero result indicates that the value is allowed to be sent to the output device.

5.2.7 System Issues

DataSafe's tag propagation is performed by the hardware logic on the *physical* memory; therefore the propagation mechanism is not changed when the protected data is passed between applications, OS components or device drivers.

Direct Memory Access (DMA) data transfers do not need to include the hardware activity-restricting tags, which are runtime tags only and are not stored in persistent storage or transmitted on a network. DataSafe treats DMA regions as output device regions and performs output control to prevent protected data (based on their hardware tags) from being written to these DMA regions. The DataSafe hypervisor also prevents SDCs from being created over allocated DMA regions (and vice versa) so that data in SDCs cannot be over-written by DMA input transfers.

The technique of checking load and store instructions for input/output violations and policy enforcement is suitable for memory-mapped I/O architecture, which is the only mechanism for the SPARC chip family since it treats accesses to the I/O space the same as it treats accesses to the memory space [72]. In other words, the communication with the I/O device registers is accomplished through memory; therefore, checking load/store instructions is enough for the SPARC architecture.

Unlike SPARC architectures, the x86 family permits direct data transfer to and from I/O ports in addition to the main memory [52]. The x86 platform uses a set of I/O instructions to access the I/O address space, in particular the family of `in`

$$\overbrace{\overbrace{\boxed{\{K_{FE}\}_{DM}}\boxed{E_{K_{FE}}(Data)}\boxed{\text{Policy}}}^{m'}\overbrace{\boxed{[hash(m')]_{HV_{Pri}}}}^{\text{OriginatorSignature}}}^{m}$$

$A \rightarrow B : m, Cert_A$
$B \rightarrow DM : \{K_{FE}\}_{DM}$
$DM \rightarrow B : \{K_{FE}\}_{HV\_B}$

Figure 5.6: Encrypted DataSafe package for storage and for transmission between machines: the originator (A), the receiver (B) and the domain manager (DM), with respective DataSafe hypervisors on A and B denoted as HV_A and HV_B. $[x]_{HV}$ denotes a private key signature or decryption operation by HV, while $\{x\}$ denotes a public-key verification or encryption operation. $Cert_A$ denotes the public key certificate of A that is signed by the domain manager.

and `out` instructions. The x86 platform can also use memory-mapped I/O where the access is handled with the processor's general-purpose move and string instructions. Our hardware enforcement of output control for Secure Data Compartments can be achieved by including the checking on the I/O instructions. Alternatively, the output checking can be performed at the processor boundary by incorporating the output control unit in the memory controller.

5.2.8 Encrypted Data Package and Key Management

A piece of data can be turned into a piece of DataSafe-protected data on any computing device within a domain that is enabled with DataSafe support. The data owner specifies the confidentiality policy for the data. We describe one implementation of key management for a domain, e.g., a hospital; many other implementations are possible. The format of a piece of DataSafe-protected data is shown in Figure 5.6. To create DataSafe-protected data that binds the owner-specified policy to the data, the hypervisor first generates a new symmetric key K_{FE}, called the file encryption key, and uses K_{FE} to encrypt the data. Each protected data file has its own random file encryption key. K_{FE} is then encrypted by the domain manager's public encryption key, K_{DM}. A domain manager is the administrator or authority that manages the computing devices within a domain and it could be installed on any DataSafe machine. The trusted DataSafe hypervisor then calculates a cryptographic hash over the encrypted K_{FE}, the encrypted data and the owner-specified policy and signs the hash using the its private signing key, HV_{Pri}, as the Originator Signature.

Transfer. Once a DataSafe self-protecting data package is created, it can be moved to any DataSafe enabled computing device within the domain for use. In a non DataSafe-enabled machine, only encrypted data can be accessed.

Unpacking. When an authorized recipient receives a piece of DataSafe-protected data and accesses it with an application, the policy/domain handler validates the data and the policy, and retrieves the file encryption key K_{FE}. Validation of the data and the policy is done by verifying that the originator signature was signed by a trusted hypervisor within the domain. A hash is re-calculated and compared with the decrypted hash in the signature, to ensure that the data, the policy and the encrypted file encryption key have not been tampered with.

Since the file encryption key K_{FE} is encrypted with the domain manager's public encryption key, the policy/domain handler follows a secure protocol to retrieve the file encryption key. The domain manager ensures that the requesting hypervisor is not on the revocation list; otherwise the request is denied.

In DataSafe, public-key crypto is used for system identification and non-repudiation to protect smaller-size items such as the K_{FE}, and efficient symmetric-key crypto is used for protecting the larger data content. Since the K_{FE} is encrypted, it is stored on the user's machine in the normal unsecured storage, whereas the hypervisor's private signing key, HV_{Sign}, and the domain manager's secret decryption key are stored in their respective DataSafe machine's hypervisor secure storage (see Figure 5.3 and Section 5.2.4 for descriptions of the hypervisor secure storage). Note that since the K_{FE} is encrypted using the domain manager's public encryption key, no key exchange between different DataSafe systems is required. Only individual communication between each DataSafe machine and the domain manager is needed (Figure 5.6).

To prevent the domain manager from becoming a bottleneck or a single point of failure, multiple or backup key management servers can be installed on other DataSafe machines to provide enhanced data availability.

Redistribution and Declassification. An authorized user can access the DataSafe protected material in plaintext, and also pass on the original DataSafe encrypted package (signed by the originator) to another machine. If he transforms the protected data and wants to pass this modified data to another machine, he has to re-package it (as described for packaging above) and sign with his own trusted hypervisor's private key.

Some data items may get declassified to be used on non-DataSafe devices. Declassification is done by the Domain/Policy Handler while the data is not in use (not loaded into memory in SDCs) by any application, and thus precludes the need to un-tag the data. This allows for authorized declassification by trusted software components – by decrypting the data, and dissociating any policy associated with it. Once declassified, such data can be treated as data that can be used on any device.

5.3 Implementation

5.3.1 DataSafe Software

Policy Handler

The policy/domain handler is primarily responsible for hardware tag generation from the high-level policy. It is also responsible for setting up the context, which includes maintaining the values for user properties, data properties, and system/environment

Listing 5.1: The structure of a sample policy in XML.

```
<?xml version="1.0" encoding="utf-8"?>
  <permissions>
    <activity num = "1" activity = "view">
      <entity-restrictions type = "Subject">
        <restriction property="Role" function="==">PrimaryPhysician</
            restriction>
      </entity-restrictions>
      <entity-restrictions type = "Environment">
        <restriction property="Location" function="==">Hospital</
            restriction>
        <restriction property="Network" function="==">Medical</
            restriction>
        <restriction property="Date" function="Between
            ">01/31/2011,12/31/2011</restriction>
      </entity-restrictions>
      <entity-restrictions type = "Resource">
        <restriction property="Part" function="include">Immunizations
            ,Medications</restriction>
      </entity-restrictions>
    </activity>
    <activity num = "1" activity = "view">
      <entity-restrictions type = "Subject">
        <restriction property="Role" function="==">Pharmacist</
            restriction>
      </entity-restrictions>
      <entity-restrictions type = "Environment">
        <restriction property="Location" function="==">Store</
            restriction>
      </entity-restrictions>
      <entity-restrictions type = "Resource">
        <restriction property="Part" function="include">Medications</
            restriction>
      </entity-restrictions>
    </activity>
  </permissions>
<?xml version="1.0" encoding="utf-8"?>
```

properties. Since both these responsibilities are specific to a particular information domain, we have a separate policy/domain handler for each domain. At present, we have implemented a policy/domain handler for Multi-level Security systems that supports a simple key protection policy, BLP confidentiality and Biba integrity policies and one for medical information systems. In all policy/domain handlers, policies are represented in the standard policy model using the XML format.

Listing 5.1 shows an example medical policy expressed in the XML format. This example policy states the following: the primay physician can view the immunization and medications records in the hospital using the medical network between 1/31/2011

Table 5.7: The policy/domain handler API.

API Call	Description
`open_file`	Open an existing DataSafe protected file.
`close_file`	Close an open DataSafe protected file.
`read_file`	Read from an open DataSafe protected file.
`write_file`	Write to an open DataSafe protected file.

and 12/31/2011. The pharmacist can view the medications record in the pharmacy store.

New policies can be specified in XML and interpreted directly by the policy interpreter. Each policy/domain handler maintains a separate database for storing user and data properties. All policy handlers share a common policy interpreter, which is possible since all policies are represented in a standard form.

File Management Module

For the prototype implementation, DataSafe software has a separate file management module that provides a file management API for accessing DataSafe-protected files and provides file handling functions, as shown in Table 5.7. This set of APIs represents the minimal set of functions that need to be redirected to DataSafe software. Other file access functions with different access granularities, such as the `readline()` function that reads a line from the file in Ruby or the `fgetc()` function that reads a character from a file in C, would also have to be included in the API to provide seamless protected-data access for an unmodified application. We list the total file access APIs that need to be included in the redirection for the C and ruby languages in Appendix A. Note again that accessing a DataSafe-protected data through an unsupported API function would simply return the encrypted data and no confidentiality policy would be violated. The file management module loads the encrypted file into the memory, and forwards the file access request to the policy/domain handler, which translates the policy associated with the file into hardware tags, and requests the hypervisor to set up SDCs for the file. Currently, the file management module supports file handling functions for Ruby and C-based applications. We first describe the Ruby implementation and then the C implementation.

We have modified the Ruby Interpreter to redirect file handling calls to the file management module. This file management module provides a file handle to the Ruby Interpreter, which it subsequently uses for file operations. If an application attempts to obtain untagged data by bypassing the redirection of file calls, it only ends up getting encrypted content. We show the file access redirection flow in Figure 5.7. When an application requests to access a file, the modified Ruby interpreter will first determine if the file is a DataSafe-protected file. If not, then the normal file `open()`

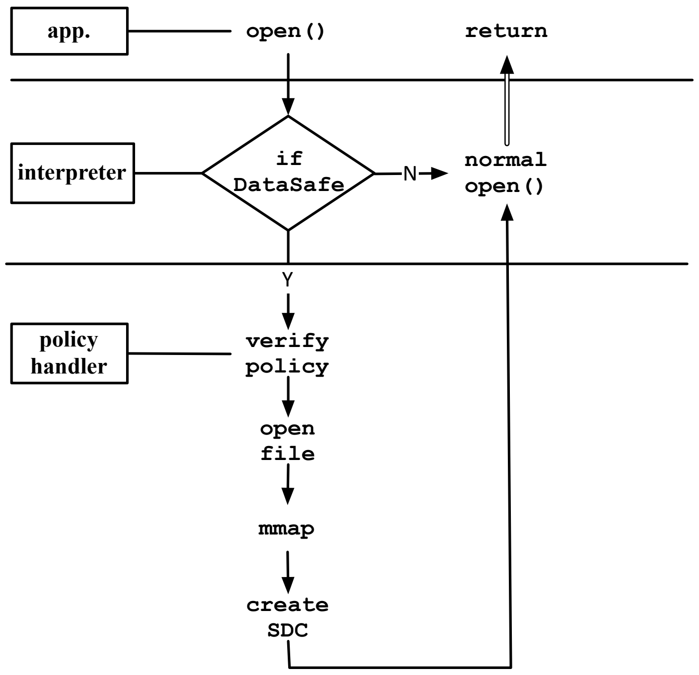


Figure 5.7: The file access redirection performed by DataSafe's file management module in the Ruby language.

function is called and no redirection is needed. If yes, then the policy handler is invoked to verify and consult the policy, open the protected file, allocate the memory space for the file (`mmap`), and then finally invoke the hypervisor to create the SDC if the access is allowed. Since the current prototype implementation does not include all of the file access-related functionalities provided by the Ruby interpreter, the file management module calls the normal file `open()` after the above steps are done by the policy handler.

A corresponding file management module for the C language is also implemented in DataSafe. However, unlike the Ruby language where the interpreter has to be modified to perform the file access redirection, we design a new dynamically loaded "shim" that interposes between the C application and the C library (`libc`) to perform the file access redirection, without having to deal with the intricate interdependencies of the C library. Figure 5.8 shows the high-level block diagram of our C shim implementation. Our C shim basically is a dynamically loaded library that is loaded before any other library (e.g., `libc`) is loaded, such that DataSafe intercepts the four basic `libc` functions for accessing a file – `fopen`, `fread`, `fwrite` and `fclose`. Similar to the flow of redirection in the modified Ruby interpreter (Figure 5.7), the C shim checks if the requested file is a DataSafe-protected file and redirects the access request to the policy/domain handler for policy evaluation and SDC creation, if the access is granted.

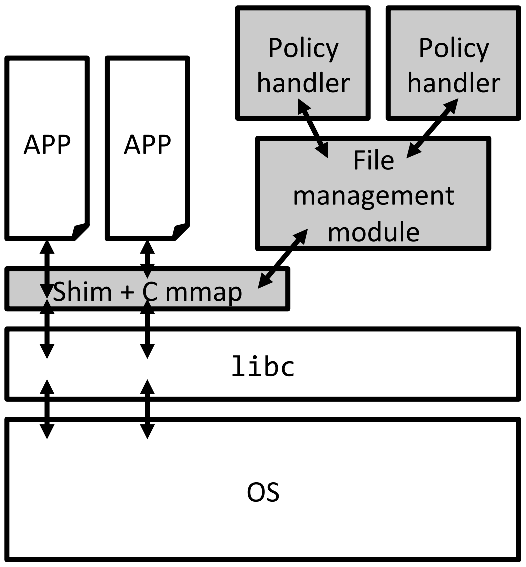


Figure 5.8: The C shim implementation for the DataSafe file access redirection.

Hypervisor

The hypervisor is responsible for the instantiations of SDCs, the management of domain-specific secret keys and the provision of environment properties for context generation. To manage the SDCs, the hypervisor keeps a software structure, called the active SDC list, `sdc_list`, which stores a list of active SDCs for all policy handlers.

Table 5.8 shows the new hypercalls introduced to support the SDCs: `sdc_add`, `sdc_del` and `sdc_extend`. Hypercalls for context generations and others are omitted. The `sdc_add` hypercall is called when the policy/domain handler requests a new SDC. The `sdc_del` is called later to delete an SDC. The `sdc_extend` is used when the high-level policy allows for appending to the protected data, where the size of a SDC is adjusted to include appended data. When the hypervisor receives the `sdc_add` hypercall with a virtual address and size of the requested SDC, the hypervisor walks the guest page table and the shadow page table to determine the corresponding physical page(s) that consist of the SDC. Multiple SDC entries are created with the same SDC ID if the virtual address range spans across physical memory ranges that are not contiguous.

5.3.2 DataSafe Prototype

Our prototype implementation builds upon the open source processor and cache hardware and the hypervisor in the OpenSPARC platform. The current prototype is implemented in the Legion simulator of the OpenSPARC platform. This simulates an industrial-grade OpenSPARC T1 Niagara processor with 256 MB of memory, running the UltraSPARC Hypervisor with Ubuntu 7.10. We utilize the load_from/store_to alternate address space (`ldxa` and `stxa`) instructions in the SPARC architecture to

Table 5.8: The new hypercalls exported by the DataSafe hypervisor.

Semantic	**Description**
`sdc_add(addr, size)`	Adds a new SDC protecting policy-encoded data starting at virtual address `addr` with size `size`
`sdc_del(sdcid)`	Deletes an existing SDC with ID = `sdcid`
`sdc_extend(sdcid, size)`	Extends an existing SDC with ID = `sdcid`, with contents of size `size`

access our new hardware structure, `mem_map`, at the same time limiting the access to only hyperprivileged software.

The open source hypervisor in the OpenSPARC platform is modified and extended with the functionality to support secure data compartments (SDCs). Our new hypercall routines are implemented in SPARC assembly and the SDC-specific functions are implemented using the C language. The policy/domain handler is implemented in the Ruby language and the policies are expressed in XML format.

5.4 Analysis

This section evaluates the security, performance and cost of the DataSafe architecture.

5.4.1 Security Tests

We tested our prototype with several experiments.

Application Independence

We tested DataSafe's capability to support *unmodified* third party applications, using three applications, Ruco, Grepper and HikiDoc, downloaded from RubyForge [11]. All three are Ruby-based applications. Ruco is a lightweight text editor, Grepper provides the same functions as the "grep" command-line utility for searching plain-text data sets for lines matching a regular expression, and HikiDoc reads text files and converts them to HTML documents. We were able to run all the three applications on DataSafe, *unmodified.* Table 5.9 summarized our security testing results.

The experiments with the Ruco editor include basic read/display and write control. In addition we modified Ruco to test illegal saving of plaintext on the disk, either with or without data transformation. A similar set of experiments were carried out with the Grepper application. In addition, with Grepper we tested fine-grained tracking by creating SDCs with different tags and sizes over different parts of a file – DataSafe could successfully track the data and enforce fine-grained output control of sensitive data.

Table 5.9: A summary of experimental results for the security testing of DataSafe.

#	Test Case	Attacks	Result
Application Independence			
1	Editor (Ruco)	read, write, output ctrl., transformation	✓
2	Search (Grepper)	read, write, output ctrl., transformation, fine-grained control	✓
3	Text Transformation (HikiDoc)	password leak (allow read but no display)	✓

With HikiDoc we tested a scenario for authorized read but prohibited display. In this scenario, simulating "password leak" attacks, the HikiDoc application takes two files as input: 1) text file (to be converted to HTML), and 2) a file containing passwords for user authentication. The program is supposed to read the password file for authentication, but not leak the password out. We inserted a malicious piece of code in the application which transforms the password into a code, and then distributes the code at predefined locations in the HTML file. The attacker can then retrieve the code parts from the HTML file, assemble the code, and reverse the transformations to get the original password. DataSafe could track the transformed pieces of a password and prevent their display.

In all these applications, the data read from the file is passed through different Ruby libraries, the Ruby Interpreter, and the operating system, before being displayed. In addition, the data is processed in different formats before being output in a presentable form. Tests on these applications show that DataSafe is application independent, can continuously track protected data after multiple transformations and can do this across multiple applications in the user space, and across the user-OS divide.

Continuous Data Tracking and Output Control

Apart from testing policy support and application independence, the experiments above also test the capability of DataSafe to enforce SDCs and hardware activity restricting tags. This includes the capability to track protected data in a fine grained manner across applications and OS, and to enforce output control only on that data which is tagged with such a restriction. The insight we derived from the above tests is that a more comprehensive, yet quick, coverage can perhaps be achieved by just a small set of synthetic test cases which represent different classes of attacks that can leak protected data, as shown in Table 5.10. In each test case, programs were run on the DataSafe machine (DS column), and on an existing non-DataSafe machine (nDS column). For each test case, the sensitive data files were protected by a policy to prohibit the test case scenario.

Table 5.10: Synthetic test suite for illegal secondary dissemination and transformation tested for DataSafe (DS) and non-DataSafe (nDS) machines. "F" represents a file, and "P" represents a program. "✗" means attack failed (good), and "✓" means attack succeeded (bad).

#	**Test Case**	**DS**	**nDS**
Output Control			
1	edit [F1, P1]	✗	✓
2	append[F1, P1]	✗	✓
3	read[F1, P1] ; save[F1, P1]	✗	✓
4	read[F1, P1] ; send[F1, P1]	✗	✓
5	read[F1, P1] ; display[F1, P1]	✗	✓
Transformations			
6	read[F1, P1] ; transform[F1, P1] ; save[F1, P1]	✗	✓
7	read[F1, P1] ; transform[F1, P1] ; send[F1, P1]	✗	✓
8	read[F1, P1] ; transform[F1, P1] ; display[F1, P1]	✗	✓
Cross-Program			
9	read[F1, P1] | save[F2, P2]	✗	✓
10	read[F1, P1] | send[F2, P2]	✗	✓
11	read[F1, P1] | display[F2, P2]	✗	✓
Transformations and Cross Program			
12	read[F1, P1] ; transform[F1, P1] | save[F2, P2]	✗	✓
13	read[F1, P1] ; transform[F1, P1] | send[F2, P2]	✗	✓
14	read[F1, P1] ; transform[F1, P1] | display[F2, P2]	✗	✓
15	***Fine-grained Transformation and Tracking***	✗	✓

Test cases 1-5 of Table 5.10 test the access type and output control capabilities of DataSafe based on output port types. In these cases, SDCs were created to prevent *edit, append, save, send over the network, and display.* Test cases 6-8 represent data transformation attacks by a single program. In these cases, a test program reads and transforms the data multiple times, and then tries to send the data out on one of the output ports (i.e. disk, network and display). Test cases 9-11 represent cross program attacks, where data is read by Program 1 (P1) and passed on to Program 2 (P2) which carries out the attack. Test cases 12-14 represent transformation and cross program combined attacks. In these test cases, data is read by Program 1(P1) and transformed multiple times, and then the transformed data is sent to Program 2 (P2), which carries out the attack. In test case 15, different parts of a file were protected by

Table 5.11: Performance costs of the C-shim DataSafe software operations vs. non-DataSafe (in cycles on the Legion simulator).

Operation	**non-DataSafe**	**DataSafe**
open	117,521.4	341,109.8
add_ sdc	N/A	10,177
read	9,016,594	2,847,026
write	2,847,026	1,659,347
delete_ sdc	N/A	3,976
close	22,076.4	278,525

SDCs with different protection tags. DataSafe was able to prevent different attacks targeting each of these protected segments. *In all the test cases, the attack succeeded in the existing machine (nDS), but DataSafe (DS) was successful in defeating the attack.*

5.4.2 Performance and Cost

Since DataSafe is a software-hardware architectural solution, its advantages come at the cost of changes in both hardware and software. These costs are in two distinct phases: 1) the *Setup* (and Termination), carried out by DataSafe software, incurs performance costs in the redirection of file calls and setting up of SDCs, and 2) the *Operation* phase, carried out by DataSafe hardware, incurs performance costs due to information flow tracking and output control. We analyze the cost of these changes separately, and then discuss the end-to-end cost of running third party applications.

Software Performance

Table 5.11 shows the costs incurred for file operations *open, add_ sdc, read, write, delete_ sdc* and *close*, for the C-shim version of file redirection. The overhead of *open* is due to file access redirection and the setting up of memory mapped regions which does not take place in non-DataSafe machines. The cost of adding and deleting SDCs on DataSafe is small compared to the other operations. These performance costs are the same for any file size.

In contrast, we actually achieve better performance during the Operation phase for *read* and *write* operations in DataSafe because of the use of memory mapped file operations. These performance gains are directly proportional to the file size (shown for reading or writing a 2.5MB file in Table 5.11). Hence, as the file size increases, the performance costs of *open* and *close*[5] get amortized leading to better results.

[5]For *close*, DataSafe software needs to delete the SDC, remove the tags and tear down the memory-mapped file, resulting in a 10X performance cost for this single operation.

Table 5.12: Performance cost (in seconds) of running Hikidoc application on increasing file sizes.

App	**0.5 MB**		**2.5 MB**	
	non-DS	DS	non-DS	DS
Hikidoc	0.53	0.67 (26.42%)	3.49	3.68 (5.44%)

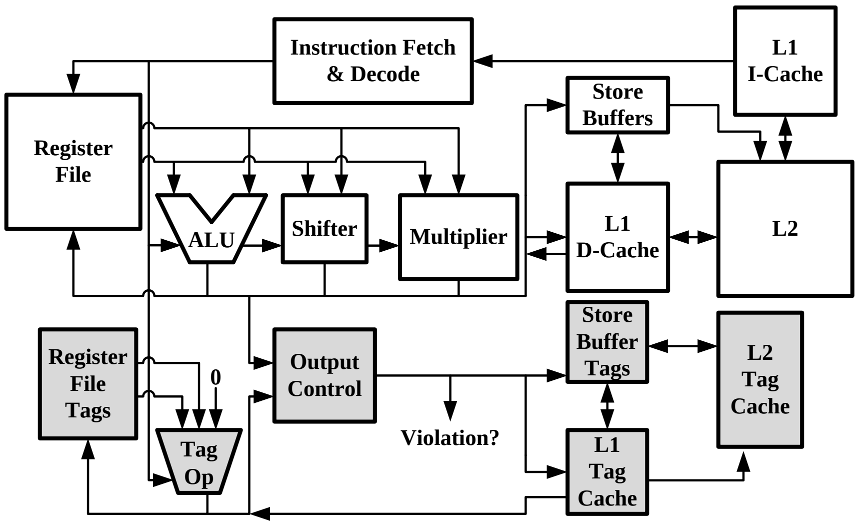


Figure 5.9: One possible implementation of the DataSafe information flow tracking processor architecture. White boxes are existing components while grey boxes are new.

This is verified by the total application execution times of different file sizes, shown in Table 5.12. As the file size increases, the relative performance cost of DataSafe decreases. For a reasonable file size of 2.5MB, the performance cost of DataSafe is only about 5%.

Hardware Performance

We now evaluate the hardware performance overhead during the Operation phase. The hardware tags can be added to the existing processor datapaths by extending the widths of the registers, buses and caches (as shown in Figure 5.5). Alternately, as shown in Figure 5.9, they can be a separate and parallel "tag datapath". This clearly shows that the tag propagation logic is done in parallel with the instruction execution, hence the hardware tag propagation does not incur runtime overhead, as also found in [34]. Comparing the hardware parallel tag datapath to the parallel execution of the software DIFT system [70], which utilizes one main application thread and one helper tag thread to process the tags, and incurs around 1.5X runtime overhead when running on a multicore system, the hardware tag provides some performance advantages that

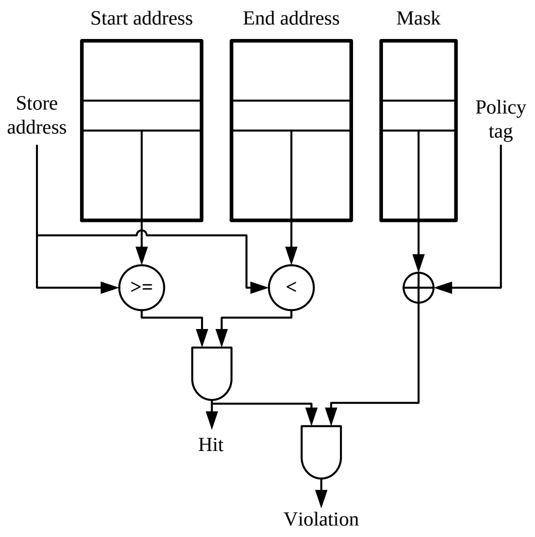


Figure 5.10: The DataSafe output control hardware component.

can be attributed to the following: (1) inter-core communication is needed for the software parallel DIFT system to communicate the appropriate flag values, and a hardware FIFO is added to [70] to speed up this communication bottleneck. On the other hand, the hardware parallel tag datapath is more tightly coupled, removing the need for this communication overhead. (2) The main application thread needs to be interrupted by the helper tag thread in the software parallel DIFT system, whereas the hardware tag datapath can produce the interrupts when violations occur, leading to a more precise interrupt and without the need for potential rollback of the system states.

Since all tag propagation operations can be done in parallel, the only source of hardware runtime overhead involves the output checking of memory store instructions. However, memory stores are not on the critical path, as opposed to memory loads, and normally stores are delayed waiting in the store buffer queue for an unused cache access cycle. Hence, the output checking can be performed while the memory store instruction sits in the store buffer.

Output control involves checking against the `mem_map` structure, similar to the operation of a victim buffer [54] or a small fully associative cache, with a different comparator design. The comparator for a victim buffer is testing for *equality*, whereas we test for *inequality*, as shown in Figure 5.10. Table 5.13 details our synthesis results to compare the cost and the access delay for the two types of comparator design, showing that they have comparable performance. This is expected since the logic designs for the two types of comparators share most of the logic gates for a magnitude comparator. A typical logic design for a 4-bit magnitude comparator [82] is shown in Figure 5.11. The comparator compares the relative magnitudes of two binary numbers, starting from the most significant bits. To achieve the inequality comparison of the start address (`>=`) for the output control, the design in Figure 5.11 would need an extra OR gate for the bottom two output signals, `A > B` and `A = B` to

Table 5.13: Comparing the cost and critical path delay of an equality comparator and an inequality comparator. The actual cost for `mem_map` would require two inequality comparators. Look Up Table (LUT) is the resource used to implement logic on an FPGA.

	Total LUTs	**MUX**	**Path delay**
Equality comparator	14	14	5.707ns
Inequality comparator	40	20	5.588ns

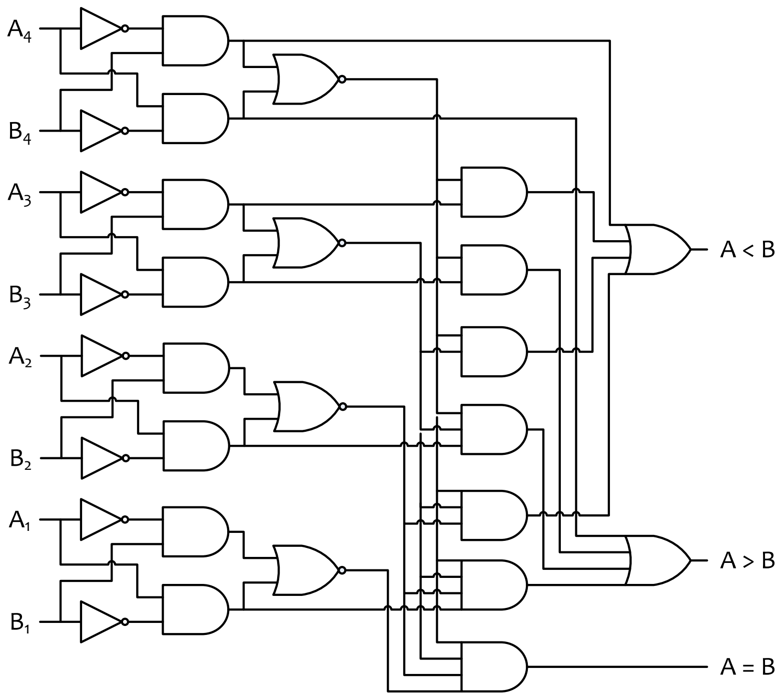


Figure 5.11: A typical logic design for a 4-bit magnitude comparator (from [82]).

get `A >= B`. However, this extra OR gate can be avoided by using the (start address - 1) as the start address. The difference in timing between the equality and inequality comparator becomes a single stage delay for an OR gate, albeit in the expense of 8 extra gates. Overall, the net effect of performing output checking on store instructions is equivalent to adding a one cycle delay for store instructions waiting in the store buffer queue. Hence, the output checking has no discernible impact on the overall processor bandwidth (in Instructions executed Per Cycle, IPC).

Storage Overhead and Complexity

The software complexity of DataSafe amounts to a total of 50% increase in the hypervisor code base, about half of which was for a suite of encryption/decryption routines for both asymmetric and symmetric crypto and cryptographic hashing algo-

Table 5.14: The complexity of DataSafe's software and hardware modules in terms of source lines of code (SLOC).

	Ruby	C
Policy/Domain handler	1197	207
Software	**SPARC Assembly**	**C**
Base hypervisor	37874	35066
DataSafe hypervisor	41657	51959
Crypto	0	16045
Hardware	**SPARC Assembly**	**C**
Base OpenSPARC	1050	51395
DataSafe hardware	0	3317

rithms (Table 5.14). Each `sdc_list` entry takes up about 26 bytes of memory space, considering a full 64-bit address space. The total storage overhead incurred by the `sdc_list` varies according to the number of entries in the `sdc_list`. In our prototype implementation 20 entries are typically used, amounting to around half a kilobyte of storage overhead.

For the DataSafe hardware, the main cost comes from the cache and memory overhead for storing the tags. For a 10-bit tag per 64-bit word used in our prototype, the storage overhead is 15.6% for the shadow memory, on-chip caches and the register file. Existing techniques for more efficient tag management [89] can be applied to reduce storage overhead. The tag storage includes 4 of the 6 new (grey) CPU components in Figure 5.9. Of the remaining 2 blocks, the Output Control block has already been described, and the Tag Operation block will be described in further detail in the next chapter.

5.5 Summary

We presented the DataSafe architecture for realizing the concept of self-protecting data. DataSafe enables owners of sensitive data to define a security policy for their encrypted data, then allow authorized users and third-party applications to decrypt and use this data, with the assurance that the data's confidentiality policy will be enforced and plaintext data will be prevented from leaking out of these authorized use sessions. Data is protected even if transformed and obfuscated, across applications and user-system transitions. Data is also protected when at-rest or in-transit by encrypted, policy-attached, DataSafe packages.

DataSafe hardware uses our enhanced dynamic information flow tracking (DIFT) mechanisms to persistently track and propagate data in-use, and to perform nonbypassable output control to prevent leaking of confidential data. Because this is done in hardware, performance overhead is minimal. However, unlike previous hardware DIFT solutions, DataSafe's key novelty is in seamlessly supporting flexible security policies expressed in software, bridging the semantic gap between software flexibility and efficient hardware-enforced policies. DataSafe is also application independent, thus supporting both legacy and new but unvetted applications. This is often a practical necessity, since users have no means to modify third-party program executables. More importantly, DataSafe provides the separation of data protection from applications, which we feel is the right architectural abstraction.

Self-protecting data, with unmodified legacy applications, may seem an unreachable goal, but we hope to have shown that it may be possible if we are willing to consider new hardware enhancements with a small trusted software base. We hope that DataSafe provides the architectural foundation over which multi-domain, multi-policy, end-to-end self-protecting data solutions can be further researched for distributed systems.

Chapter 6

Practical Information Flow Tracking

From Chapter 5, we see the importance of *tracking* data after an untrusted application is granted access to the sensitive data. We go into the details of how this can be achieved in this chapter. We use a technique called Information Flow Tracking (IFT), where we first show the different ways information can flow within a system. We propose a taxonomy of various information flows that can occur while a program is executing and discuss the practical issues of a particular kind of IFT called Dynamic Information Flow Tracking (DIFT), on which we base our solutions. Then, we present the prior work using this technique. Finally, our two different approaches to address the issues of DIFT are described and we show experimental results that demonstrate the practicality of our proposed information flow tracking systems and how information flow tracking can help prevent information leakage and protect data confidentiality.

In summary, this chapter makes the following contributions:

- a taxonomy of information flow in computer systems,
- analyzing and proposing different hardware solutions for solving the implicit information flow problem, especially reducing false positives,
- practical hybrid solutions using binary code analysis to inform hardware DIFT schemes, providing strong security (no false-negatives) and usability (low false-positives) guarantees, and
- practical hardware-only solution that uses register save and restore mechanism to prevent unnecessary propagation of tags between application and the operating system space to reduce false positives.

6.1 Background of Information Flow Tracking

Digital information (or data[1]) is an important asset, e.g., a person's social security number or medical record. On the other hand, the information can also be used as an effective weapon. For example, malicious or malformed input can cause buffer overflow, which may potentially lead to a complete compromise of system security. Furthermore, information can be abused or misused to either carry information that

[1]For the purpose of this chapter, we use *information* and *data* interchangeably.

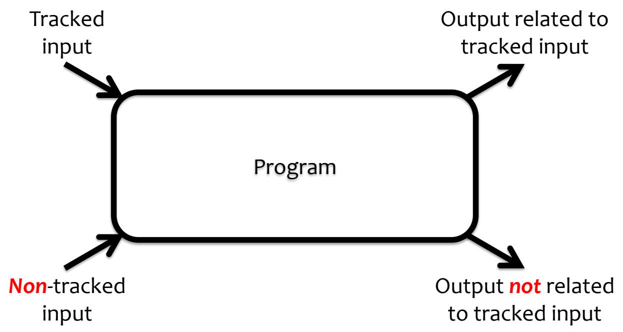

Tracked input
Output related to tracked input
Program
Non-tracked input
Output not related to tracked input

Figure 6.1: A high-level view of tracking information flow in computer programs. The goal is to be able to distinguish whether or not the program's output is related to the input that we wish to track.

is not intended, i.e., covert channels, or to gather information indirectly, i.e., side channels. Although confidentiality and integrity can often be considered duals of each other, in this thesis, we focus on protecting the confidentiality of data. In other words, we treat information as an asset and want to prevent information leakage using IFT. In this section, we identify and categorize different possible information flows that could occur within a computer system. In particular, we focus on the aspect of information flow pertaining to the *execution* of the untrusted application.

Figure 6.1 summarizes the high-level view of tracking information flows during program execution. A program's input can be divided into two parts – tracked input, i.e., the sensitive data we wish to track, and non-tracked input that is not related to the sensitive data, e.g., constants or environment variables. The goal of information flow tracking is to be able to identify which part of the program's output is related to the tracked input, and which part is not, in order to determine if the program is leaking any sensitive information. A program's execution consists of the execution of a chain of instructions. Therefore, in order to understand how information flows or propagates through the execution of a program, we look at how information flows within an instruction and across instructions.

Before we look at the details of each machine instruction, here are the basic rules or assumptions of IFT:

1. Each data storage location is extended with an associated data "tag", e.g., memory, cache lines, registers or entries in a load-store queue.
2. The tag is considered as attached to the data and travels along with the data wherever the data is stored or processed.
3. The tag can be of arbitrary size, representing different meanings depending on the particular architecture.
4. For the purpose of this thesis, a non-zero tag value denotes the corresponding data to be sensitive and protected, whereas a zero tag value denotes non-sensitive data.

Information flows resulting from the execution of program instructions can be largely divided into two categories: (1) explicit information flows, and (2) implicit information flows. Explicit information flows refer to the cases where actual data values are propagated from the source to the destination. For example, a `load` instruction copies the data value from the memory (source) to the register (destination). Correspondingly, the tag of the same memory location should be copied to the tag of the destination register, to capture the information flow from the memory to the register. On the other hand, implicit information flows occur not by direct data copying, but through program side-effects. For example, if the time taken to execute an instruction is dependent on a tagged register value[2], although there is no direct data movement between the tagged register and the time measurement, the time measurement does carry information related to the tagged register value. We consider these implicit information flows as side-channels or covert-channels. Figure 6.2 shows the taxonomy of information flows resulting from the execution of a program. Before we give examples for each of the items in Figure 6.2, we sketch out the skeleton for an ideal information flow tracking system as follows:

1. An ideal information flow tracking system should exhibit no false positives and no false negatives. In other words, an ideal system should be able to distinguish between data that is truly related to the source input and data that is not related.
2. An ideal information flow tracking system should be able to recognize special conditions that erase information, such as zeroing a register by `xor` with itself.
3. An ideal information flow tracking system should be able to track seemingly unrelated events, e.g., timing or power, and determine if those output channel contain information related to the source input.
4. An ideal information flow tracking system should be able to identify data that are truly conditionally dependent on the source input, and track the information flows correctly. For example, a data value that is modified when a certain condition happens should be tagged, whereas a variable that happens to be assigned the same value under all possible conditions should not be tagged.
5. An ideal information flow tracking system should be able to understand program semantics to determine whether or not information flows across table lookups. For example, character conversion using table lookups does carry information from the table lookup index, whereas information should not propagate from node to node in a linked-list traversal, if only one of the node is tagged.

6.1.1 Explicit Information Flow

- **Arithmetic**: an arithmetic operation such as `add`, `sub` or `div` usually takes two source variables, computes the result and puts the result into the destination variable. For example, an assignment $a \leftarrow b + c$ means that the variable a gets information from both variables b and c. Correspondingly, we denote the tag assignment to be $tag(a) \leftarrow join(tag(b), tag(c))$. The $join()$ operation is definable according to how the tags are defined. For an IFT scheme with 1-bit

[2]We consider a data "tagged" if its corresponding tag value is non-zero.

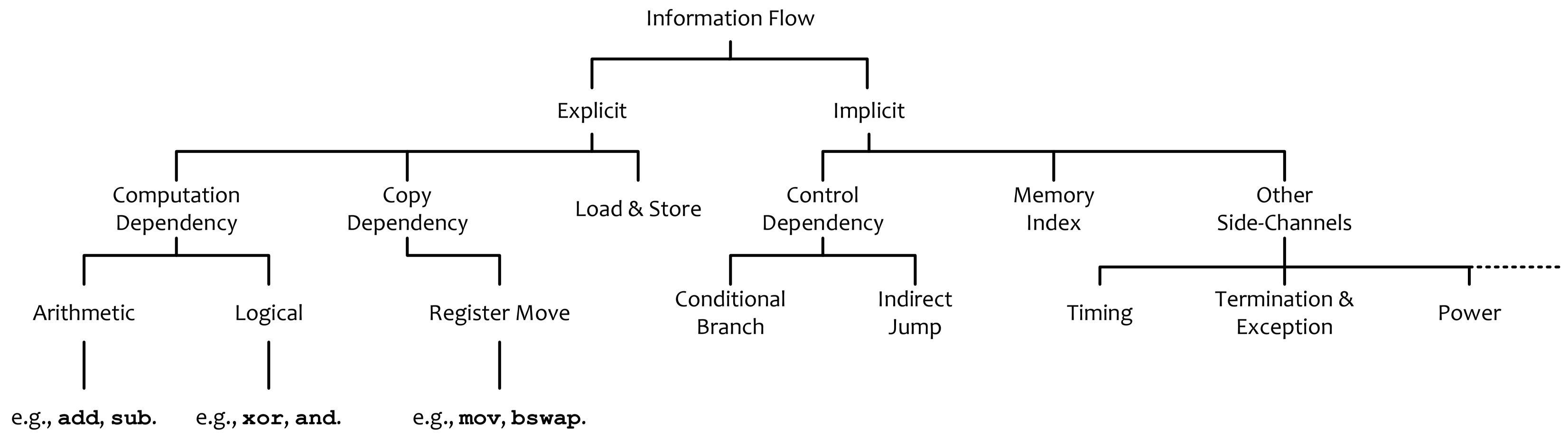


Figure 6.2: A taxonomy of information flows within a program.

tags, the *join*() operation can be defined as a simple logical `or` of *tag*(*b*) and *tag*(*c*).

- **Logical**: a logical operation such as `or`, `and` or `xor` behaves similarly to arithmetic instructions described above. Therefore, they use the same *join*() operation for the tags. Special cases such as zeroing a register using `xor`s are treated differently, since no information flows from the source registers to the destination register in these cases.
- **Memory Loads & Stores**: `load` instructions copy data values from the main memory to the destination register, whereas `store` instructions copy data values from the source register to the main memory. Correspondingly, the tag of the source (memory or register) is copied to the tag of the destination as well. For a `load` example, $a \leftarrow [addr]$, the tag assignment should be $tag(a) \leftarrow tag([addr])$.

The above examples are cases where direct data flows are involved within an instruction. Next we give examples where information flows without having direct data flows.

6.1.2 Implicit Information Flow

- **Conditional Control Dependency**: modern general purpose computer architectures employ certain mechanisms to conditionally execute some code, which is often realized using a set of status or flag registers, e.g., the `FLAGS` register in x86 or the Condition Code Register (`CCR`) in SPARC. A typical set of flags include the zero flag, the carry flag, the sign flag and the overflow flag. An arithmetic or logical instruction can affect the value of these condition flags if certain condition is met. For example, the `if` statement in C can be implemented using an arithmetic instruction that sets the condition flag with a branch instruction that branch to certain locations in the code depending on the value of the condition flag. Therefore, it is possible for information to flow from the arithmetic instruction to the condition flag, and from the condition flag to the instructions executed after the branch instruction. In this thesis, we focus on this case of implicit information flow and will go into the details in Section 6.1.3.
- **Indirect Jump**: indirect jumps execute some code based on the jump target value stored in a register (or sometimes, in two registers). Therefore, information flows from the register that stores the target address to the instruction executed after the jump instruction. If the jump targets can be known or determined statically before the program's execution, we can handle the indirect jumps in a similar fashion as conditional branches, since they both set the new program counter's value depending on the value(s) of some register(s). However, if the jump targets cannot be determined statically, the entire memory space could potentially be the target for the indirect jump. Program-level information is needed to determine whether or not there is information flow across the indirect jump.
- **Memory Indexing**: memory indexing is related to the memory loads and stores described previously, regarding the *addr* component that is used to decide

which memory address to read from or write to for a `load` or `store` instruction, respectively. The fact that the *addr* component decides the memory address implies that *some* information is passed from the *addr* to the destination, whether the destination is a memory location or a register. Therefore, taking the memory indexing into account, we have a modified tag assignment for the previous `load` instruction example: $tag(a) \leftarrow join(tag([addr]), tag(Reg_{addr}))$, where Reg_{addr} denotes the register that contains the value of $addr$. For the previous `store` instruction example: $tag([addr]) \leftarrow join(tag(a), tag(Reg_{addr}))$, where Reg_{addr} denotes the register that contains the value of $addr$. To properly address the issue stemming from memory indexing, we need to analyze the intended operations of the source code. For instance, some keyboard character mapping routines use table lookups [99], and thus tag propagation should be performed in this case to avoid false-negatives for key-logging malware. However, in some cases a table lookup result would provide a sanitized value that does not need to be tagged. Since this chapter focuses on the case where only the application binary is readily accessible and no source code is available, we do not take into account the memory indexing issue in this chapter, assuming that no information flows from the index to the destination.

- **Timing**: we can gather information about a certain operation without direct data copying, just by measuring the amount of time taken for the operation. Consider the following example, adapted from [20]:

```
const int x = 1;

send_value(adversary_site, x);
y = ...sensitive...

... // delay loop
... // time propotional to y

send_value(adversary_site, x);
```

  In this example, the adversary can gain information about y by measuring the time difference between the two `send_value` operations, even though only a constant value x is sent to the adversary. Ideally, a correct IFT scheme should be able to catch this with a tag assignment of $tag(x) \leftarrow tag(y)$.
- **Termination & Exception**: besides normal program execution, abnormal program execution can also be leveraged to gain information about the state of a program. Consider the following example, adapted from [94]:

```
for (i = 0; i < MAX; i++) {
    if (i == secret) {
        throw_exception();
    }
    printf("x");
}
```

Assuming that the `secret` value is 5 and that `MAX` is larger than 5, then the execution of this particular example will output "`xxxxx` before the program terminates with the exception (`throw_exception()`). Although the `printf` statement is not dependent on the `secret` value and thus does not contain information related to the `secret` value, the occurrence of the exception changed this assumption by terminating the execution before the end of the program.

- **Other Covert- and Side-Channels**: besides the methods of gathering information without direct data copying as described, there are several other ways to gain information indirectly about certain operations in a computer systems. Our listing is not exhaustive and, as new technology evolves, new side-channels will be exploited to convey information implicitly. Examples of other side-channels include probabilistic [26], resource exhaustion[3], and power [57] side-channels.

6.1.3 Information Flow Through Conditional Execution

Addressing implicit information flows stemming from memory indexing [86], timing or other side-channels, typically requires higher-level knowledge of the program semantics, thereby requiring access to the program's source code. In this chapter, we deal with third-party application binaries, focusing on the case of the condition flag in the implicit information flow, as it occurs most often in legitimate code, e.g., format conversion, and is prevalent in document editing programs [55]. Although it is one of the most common cases of implicit information flow, it is difficult to correctly track using IFT schemes.

The following code snippet shows a simple example of implicit information flow through the condition flag dependency of a program:

```
x := y := 1
if s <= 0 then x := 0 else y := 0
```

There clearly is a flow of information from the variable s to x; however, it is not the result of direct data copying, but the result of affecting the branch outcome by setting the condition flag through the comparison. For this particular example, an IFT scheme may have the tag assignment $tag(x) \leftarrow tag(s)$ when $s \leq 0$ and $tag(y) \leftarrow tag(s)$ when $s > 0$. However, if we look closely, we can conclude that both tag assignments for x and y should *always* be performed, no matter the value of s, since observing the value of either x or y would reveal information about the assignment that *did not* occur, and we can use this information to reveal the signedness of s. Before we describe our proposed solutions to address this kind of implicit information flow, we look at various prior IFT schemes to understand what is needed for a practical information flow tracking system.

[3]A value of 0 or 1 could be interpreted by the availability of a specific resource which may be filled up (hard disk), overloaded (100% cpu), etc.

6.2 Prior Work

Information flow tracking can be achieved either statically before the program is run, dynamically when the program is running, or both. In addition, the tracking can be done at the software level or at the hardware level. The granularity of the tracking can also be varied depending on the application, e.g., at the lowest gate level or at a higher operating system objects level. We review different IFT systems, regardless of whether their original intent was to protect integrity or to protect confidentiality.

Static language-based software techniques [81] track information by type-checking programs that are written in languages that express information flow directly. Programmers can specify the legitimate information flows and policies in the program such that no illegal information flow would be allowed once the program is compiled. This static method can be formally verified to be secure and can address most of the implicit information flow and side-channels since the high-level semantics of the program can be checked. However, it requires access to the source code, requires re-writing or re-compiling the applications and makes the programmer responsible for specifying the information flow policy.

On the other hand, information flow tracking can be done dynamically, which is usually called dynamic information flow tracking (DIFT). Different tracking granularity can be applied when tracking information dynamically in the software. At a coarse-grained level, new operating system designs like HiStar [101] and Asbestos [39, 40, 96] proposed labeling of system objects to control information flow. In these systems, a process (thread) that has accessed tagged data is not allowed to send any data to the network, even if the data sent has no relation at all to the tagged data. This coarse-grained information flow protection requires the application to be partitioned into components with different privilege levels. Tracking information flow dynamically at a finer-grained level such as bytes can be achieved using binary translation [77]. However, such software-only DIFT approaches often incur prohibitive performance overhead [77]. For example, to deal with tag assignments and bookkeeping, a single data movement instruction becomes eight instructions after binary translation. A single arithmetic/logic or control flow instruction is replaced by 20 instructions after binary translation. Even with parallel execution of the binary translation [70] the performance overhead is around 1.5X.

Tracking information flow dynamically at the byte or word level can be done a lot more efficiently at the hardware level, since often times the tag assignment can be done in parallel to the instruction's execution. Raksha [34] is a hardware DIFT system which can detect both high-level and low-level software vulnerabilities for integrity purposes. Raksha exposes software configurable registers to support four security policies at a time. However, since Raksha is targeted for integrity applications, it does not deal with implicit information flow or side-channel problems.

With hardware support, information tracking can be done at a even finer-grained level. GLIFT [92] is another hardware DIFT solution that tracks information flow at a much lower hardware level – the gate level. It uses a predicated architecture[4] which executes all paths of a program (i.e., no path is untaken) to track both explicit and implicit information flow due to condition flags, albeit at the cost of execution efficiency. While this is a very interesting and potentially promising approach, all the hardware has to be re-designed from the gates up, requiring unproven new hardware design methodologies and tools. Furthermore, it requires re-writing or re-compiling existing applications, making its practical adoption even more difficult.

Suh et al [89] proposed the architectural support for DIFT to track I/O *inputs* and monitor their use for integrity protection, whereas our methods provide support for DIFT to track I/O *outputs* and monitor the use of confidentiality-protected data. They assume that the programs can be buggy and contain vulnerabilities, but they are not malicious and that the OS manages the protection and is thus trusted. One bit is used as the security tag that indicates whether the corresponding data block is authentic or spurious. The proposed technique also does not track any form of control dependency; in other words, implicit flows are not tracked, since the authors believe that it is difficult for attacks to exploit control dependencies because programs do not use control dependencies to generate pointers in the authors' set of benchmarks. The authors assume that adding non-constant offset to the base pointer implies that the program performed a bounds-check before. The authors do acknowledge that not propagating the tags through table lookups or pointer additions may be problematic; however, they did not present any false negatives in their experiments.

To be able to track implicit information flow while incurring minimal performance overhead, a hybrid approach that combines static analysis with DIFT is desirable. RIFLE [94] is a hybrid approach that uses compiler-assisted binary re-writing to change the program to turn implicit information flows due to condition flags into explicit tag assignments. Once the implicit flows are turned explicit, the hardware can track these explicit information flows efficiently. The BitBlaze project [87] also combines static binary analysis and dynamic tracking for application binaries for various purposes, e.g., spyware analysis [41, 99] and vulnerability discovery [18, 32]. Note that this hybrid approach is not to be confused with combining *static* information flow tracking with DIFT, as the static analysis does not track the information flow but merely assists the DIFT scheme to provide information which may be missed for a pure DIFT scheme.

Cavallaro et al. [21] discussed several practical examples that can be used by malware to evade dynamic taint analysis. For example, control dependence, pointer indirection, implicit flows and timing-based attacks are desribed. More concrete examples are explained in the context of browser helper objects (BHOs), which can be categorized as plug-ins in some browser architectures.

[4]In a predicated architecture, the effect of an instruction is guarded by a predicate register. The operations for both cases (predicate true/false) get executed, but only the instructions whose predicates evaluate to true actually write their value back to a register.

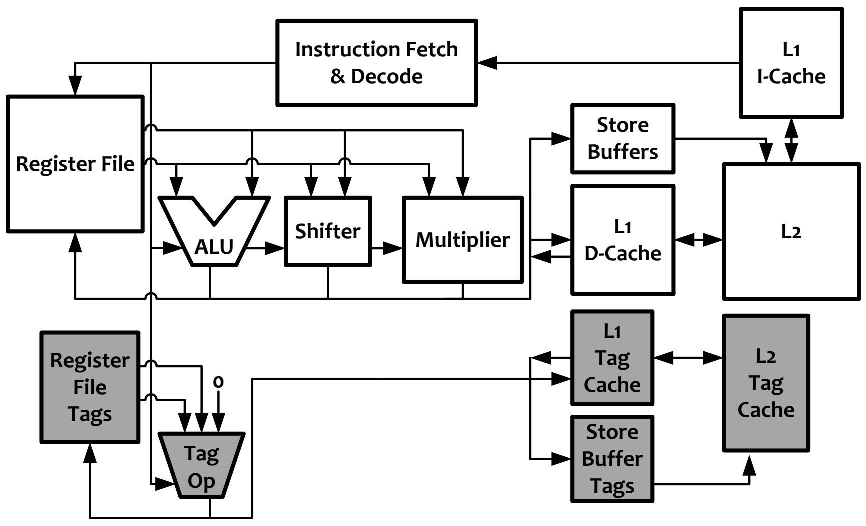


Figure 6.3: The baseline hardware DIFT architecture. The white blocks are basic processor components whereas the gray blocks are DIFT components.

6.3 Mitigation Techniques for Practical Dynamic Information Flow Tracking

In this section, we propose a few hardware or hybrid mitigation techniques to address the issues of false-positives (FPs) and false-negatives (FNs) of the DIFT system and describe their concept and implementation. The security, complexity and performance evaluation are given in Section 6.4.

This chapter focuses on the confidentiality aspect of the DIFT system, and the techniques presented in this section aim to address implicit information flow and the FPs and FNs due to control dependencies. In order to track control dependencies dynamically, we first establish a baseline DIFT system (as shown in Figure 6.3) with such a capability to reduce FNs arising from control dependencies. Let us revisit the conditional execution example given in Section 6.1.3 and see how the information flow problem is solved using static information flow tracking and whether or not we can adopt the same technique dynamically:

$$\textsf{if } s <= 0 \textsf{ then } x := 0 \textsf{ else } y := 0$$

Language-based static information flow tracking techniques [81] solve this problem by introducing the concept of the *program counter tag*, denoted *Tpc*, which indicates the information that can be learned by knowing the control flow path that has been taken in the program. In the case of the above example, since the taken path depends on the value of the variable s, the *Tpc* in the `then` and `else` clauses will inherit the label of s. If the source code of the program was available, this can be more easily achieved by the compiler performing the static analysis; thus the program would contain no illegal information flow if it passes the compilation.

We can apply the same technique to dynamic information flow tracking to track information across conditional execution. Suppose we have Tpc that gets the tag of s dynamically and every instruction takes into account the value of Tpc in addition to the tags of the source registers or memory locations. We will be able to correctly capture the information flow from s to either x or y, even though they are assigned a constant value in either path. Unfortunately, without knowing the scope of the branch instruction dynamically, false positives[5] may arise depending on when the Tpc is cleared. A naïve DIFT system that tracks control dependency might tag every instruction that is executed after the tagged branch, thus leading to an unusable system with a large amount of false positives.

6.3.1 Tunable Propagation Tracking for Control Dependency

As a simple mitigation technique for reducing the amount of false positives in the naïve DIFT system, we utilize a count-down counter Tpc_CNT that counts from a maximum value, $FLOW\_MAX$, to zero, to determine when to clear the Tpc. The counter value is set to the maximum value whenever there is a branch instruction that sets the Tpc to a non-zero value, indicating a tagged conditional execution. Tpc_CNT decreases by one after each instruction is executed when the counter value is greater than zero and Tpc is reset (set to zero) when the counter Tpc_CNT reaches zero. Tpc_CNT does not go below zero. The process starts again when the program's execution reaches another tagged branch instruction. We call Tpc_CNT the propagation counter since it determines how much information is propagated due to a tagged branch instruction. From a different perspective, clearing the Tpc_CNT can be described as *declassification* in the program, since clearing the Tpc essentially declassifies the sensitivity of the data that are operated upon by the instruction.

This propagation counter technique requires adding one 32-bit count-down counter to the baseline architecture and one 32-bit register to store the value of $FLOW\_MAX$.

6.3.2 Using Static Analysis to Reduce False Positives

The simple tunable propagation technique described in the previous section requires the appropriate setting for the $FLOW\_MAX$ value for a particular application. However, this is often a very hard decision to make since setting the value too low would create false negatives that eventually leak information, and setting the value too high would create false positives just like in the naïve DIFT system.

Since it is difficult to determine the optimal $FLOW\_MAX$ value dynamically, we resort to static techniques that can assist the DIFT system in determining the optimal value. For this section, we look at a similar conditional execution example

[5]False positive (FP) is defined as the variable or memory location that is tagged while it contains no information related to the source tagged data. False negative (FN) is defined as the variable or memory location that is *not* tagged while it *does* contain information related to the source tagged data.

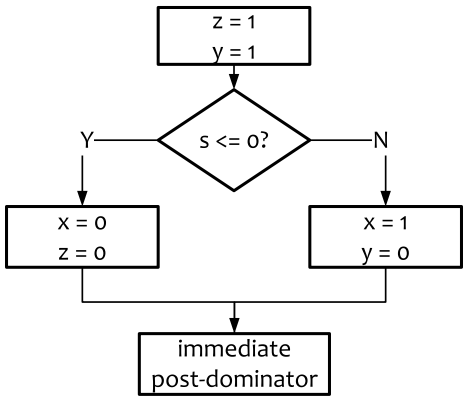


Figure 6.4: The control flow graph (CFG) of the conditional execution example. A node p post-dominates a node a if all paths to the exit node of the graph starting at a must go through p.

as before but slightly more complicated to see how static analysis can help a naïve DIFT system.

$z := 1;\ y := 1;$
$\mathsf{if}\ s <= 0\ \mathsf{then}\ x := 0;\ z := 0;\ \mathsf{else}\ x := 1;\ y := 0;$

The control flow graph of this example is shown in Figure 6.4. From the control flow graph, we can determine that the information should flow and be tracked from the tagged branch block (`s<=0?`) until the start of its immediate post-dominator block, where declassification should happen. Intuitively this means that we are allowing the Tpc to affect only the instructions that are control-dependent on the tagged branch but nothing more. In practice, this static analysis can help the tunable propagation in two ways. The first one is to run the static analysis to figure out the appropriate $FLOW\_MAX$ value for each potentially tagged branch instruction and insert an extra instruction before the branch instruction to set the $FLOW\_MAX$ value, in order to guide the DIFT system to reduce the amount of false positives. An alternative implementation is to set the $FLOW\_MAX$ value to a large enough value that fits the desired security requirement, run the static analysis and insert an instruction at the start of the immediate post-dominator block to clear the Tpc. This implies that now we do not have to determine the optimal value of $FLOW\_MAX$, but can set the $FLOW\_MAX$ arbitrarily while reducing false positives due to tagged conditional branches.

In practice, clearing out the Tpc at the immediate post-dominator block only works for a single level of tagged branch. To account for potentially nested levels of tagged branches, we employ a Tpc stack[6] to push the old Tpc value at the tagged branch block. We will describe Algorithm 3 which shows how to insert pushes and pops into this Tpc stack. Complications dues to loops are also illustrated with Figures 6.5 and 6.6.

[6]The Tpc stack lies in the protected memory space of the hypervisor (Chapter 5).

Algorithm 3: Instrument a program to reduce false positives due to tagged branches.

```
Input: Set of basic blocks
Output: Instrumented program binary
1  foreach tagged branch block $t$ do
2      Find $t$'s immediate post-dominator block $p$;
3      Insert an empty block to ensure a unique immediate post dominator block,
       if necessary;
4      if $p$ is inside a loop and $t$ is outside then
5          Do a loop peeling to make sure $t$ and $p$ are at the same loop level;
6      end
7  end
8  foreach tagged branch block $t$ do
9      Find $t$'s immediate post-dominator block $p$;
10     Push $Tpc$ onto stack: push($Tpc$);
11     foreach path from $t$ to $p$ do
12         if $t$ reaches itself before reaching $p$ then
13             Insert in the path before reaching back to $t$, if a pop has not been
               inserted there for $t$: Pop old $Tpc$ back from stack, $Tpc$ = pop();
14         end
15     end
16     Insert before the beginning of $p$: Pop old $Tpc$ back from stack: $Tpc$ =
       pop();
17 end
```

Since nested branches may share the same post-dominator block, we insert empty blocks to make sure that each tagged branch block has a unique immediate post dominator block (line 3 of Algorithm 3). Lines 4 to 6 shows the loop-peeling process to prevent the *Tpc* stack from being out-of-sync due to the first iteration of a loop. Additionally, to handle loops that may reach the same node again before reaching the post-dominator block of a tagged branch, our algorithm detects if the same node is visited again and inserts a pop operation right before the code reaches the same node (lines 11 to 15 of Algorithm 3). These mechanisms ensure that each push operation has a corresponding pop operation. Algorithm 3 sketches out the static binary analysis and instrumentation. Note that this type of static analysis can be done on the program binaries [87], and does not need the source code.

Figure 6.5 and 6.6 give an example for the loop peeling process in Algorithm 3 (lines 4 to 6). The loop peeling process is necessary to correctly match the number of pushes and pops of the *Tpc* stack, such that the stack is not out-of-sync with the program control flow. An example control flow graph is shown in Figure 6.5, with a loop in between node `B` and `J`. Loop peeling works by first duplicating all the nodes and edges within the loop, nodes `B’` to `J’` in Figure 6.6. Then the loop back edge

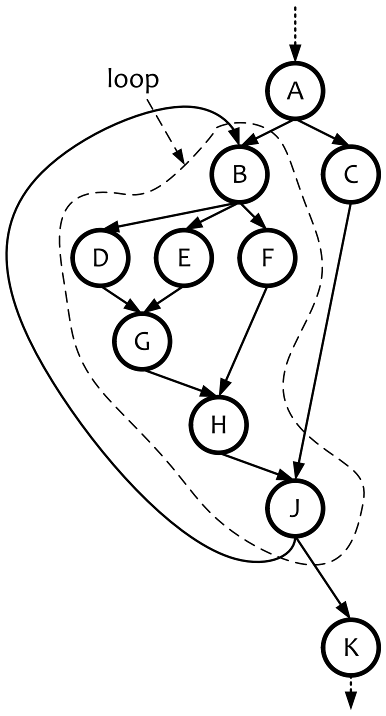


Figure 6.5: An example control flow graph to illustrate the loop peeling of Algorithm 3.

from `J` to `B` is diverted to `B'`, and the exit edge from `J'` to `K` is added for the peeled loop.

The main purpose of loop peeling is to distinguish between the first loop iteration and the rest of the iterations. Taking the tagged branch block `A` in Figure 6.5 for example, `A`'s immediate post dominator block `J` is within the loop. If we perform Algorithm 3 without the loop peeling part, we would instrument the program to push the *Tpc* at block `A` and pop the *Tpc* before the block `J` (at lines 10 and 16, respectively).. However, since `J` is within the loop, there will be many more pops that do not have corresponding pushes, causing the stack to be out-of-sync with the program's execution. After the loop peeling, `A`'s immediate post dominator block `J`, is now *not* in the loop and only executed once, so there will only be a single pop for the push of *Tpc* in block `A`.

As another example, we focus on the block `J` in Figure 6.5. If we perform Algorithm 3 without the loop peeling part, we would push the *Tpc* at block `J` and insert a corresponding pop before reaching `J` in each path leading back to `J`, i.e., `H` to `J`. However, if we consider the first time the loop is entered, the pop operation actually happens before the push at block `J`, leading to an out-of-sync *Tpc* stack. Therefore, by breaking the loop for the first loop iteration using loop peeling, we would insert the pop in the path `H'` to `J'` instead, so the pushes and pops for the first iteration of the loop will not result in a out-of-sync *Tpc* stack.

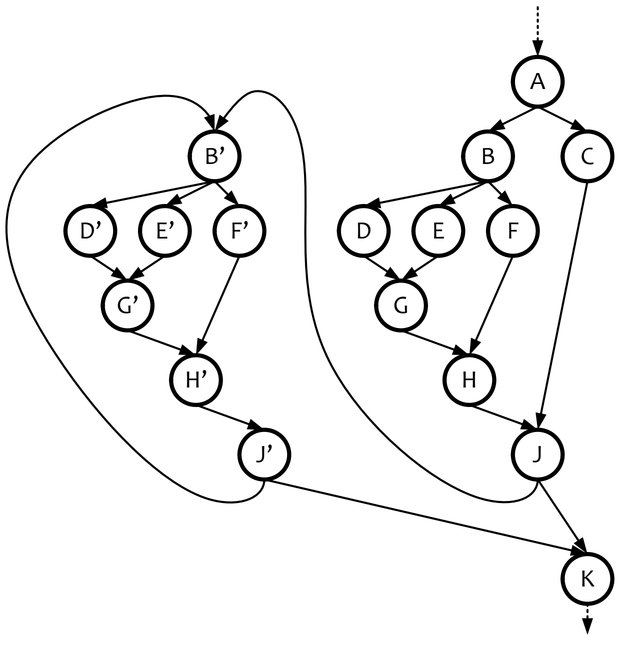


Figure 6.6: The modified control flow graph after loop peeling of Algorithm 3.

6.3.3 Using Static Analysis to Reduce False Negatives with Untaken Paths

Another issue remains even though a near optimal value of $FLOW\_MAX$ can be determined. Let's refer back to the code example used in the previous section. Suppose we use the DIFT scheme in a Multi-Level Security (MLS) system where x has no security tag, s and y are tagged `HIGH` while z is `LOW`, and we want to prevent information flowing from `HIGH` to `LOW`. When s is less than or equal to zero, a violation would be detected since information flow from s to z is not allowed. However, when s is greater than zero, no violation occurs. Nevertheless, an adversary can still gain information about the value of s just by observing the value of z, since z was assigned a value 1 before the `if` statement. This problem stems from the fact that only one path of the program is taken dynamically, so the Tpc does not affect the tag of the variables that are supposed to be modified by the path that was *not* taken. To solve this problem dynamically, we would need to compensate for the tag assignments in the untaken path, as shown in Figure 6.7.

Figure 6.7 shows the tag assignment compensations that account for the untaken path. For the left path, we insert the tag assignment for y such that although the $y = 0$ statement is not executed, we can still ensure that the implicit information flow is tracked dynamically. The same holds true for the right path in Figure 6.7. To enhance Algorithm 3 to achieve the effect of Figure 6.7, we need to identify the variables or memory locations that are changed between the tagged branch block and its immediate post-dominator block. This set of variable and memory locations can be identified using standard compiler analysis and we omit the details here.

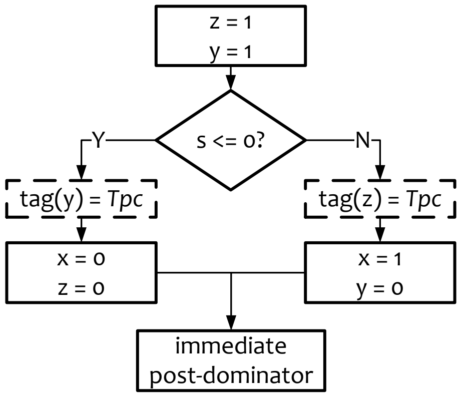


Figure 6.7: The instrumented control flow graph of the conditional execution example after the compensation for tag assignments to account for the untaken path in the dashed boxes.

Algorithm 4 assigns proper tag assignments to the variables for the untaken path[7]. *succ*[*n*] denotes the successor blocks of *n*. These extra tag assignments are inserted for each path, since each path may have touched a different set of variables. We only need to find the variables that are *live* at the immediate post dominator block and have been assigned different values in between the tagged branch block and the post dominator block (*vset*), since data that is not *live* will not be able to leak any information, and data that is not changed between the tagged branch block and the post dominator block does not carry any information regarding the tagged branch. Since during program execution, variables that are assigned a value between the tagged branch block and the post dominator block (*cset*) will already be tagged with the *Tpc*, we do not need to duplicate their tag assignment. Although the algorithm may insert some redundant tag assignments, late phase of optimizations, such as dead code and redundancy eliminations, will clean up the code. It addresses the issues of false-positives and false-negatives for implicit information flows due to conditional execution and the untaken path.

Nevertheless, there is a limitation regarding *live* memory locations. If the address of the live memory location can be determined statically, the algorithm can instrument the proper tag assignment for the untaken branch. If the address cannot be determined or not all of the target memory locations can be identified statically, we need some high-level program information to help guide the static analysis about the potential live memory addresses, e.g., by annotating the base and bounds of a table used in a table lookup.

[7]The analysis includes the condition code register (`CCR`), although the individual `CCR` register flags are usually used implicitly, e.g., through compare instructions, and applications typically do not access the `CCR` register directly. However, a compromised application could include code that specifically tests the value of the `CCR` register; therefore the tag of the `CCR` register also needs to be accounted for on the untaken path.

Algorithm 4: Instruments a program to reduce false negatives with untaken paths.

```
Input: Set of basic blocks
Output: Instrumented program binary
foreach tagged branch block $t$ do
    Find $t$'s immediate post-dominator block $p$;
    Insert an empty block to ensure a unique immediate post dominator block,
    if necessary;
    if $p$ is inside a loop and $t$ is outside then
        Do a loop peeling to make sure $t$ and $p$ are at the same loop level;
    end
end
foreach tagged branch block $t$ do
    Find $t$'s immediate post-dominator block $p$;
    Push $Tpc$ onto stack: push($Tpc$);
    foreach path from $t$ to $p$ do
        if $t$ reaches itself before reaching $p$ then
            Insert in the path before reaching back to $t$, if a pop has not been
            inserted there for $t$: Pop old $Tpc$ back from stack, $Tpc$ = pop();
        end
    end
    Find $vset$ = variables and memory locations that are changed between $t$
    and $p$, and reach $p$;
    foreach $j \in succ[t]$ do
        Find $cset$ = variables and memory locations that are defined between $j$
        and $p$;
        foreach $m \in (vset - cset)$ do
            Insert tag assignment in the beginning of $j$: tag($m$) = $Tpc$;
        end
    end
    Insert before the beginning of $p$: Pop old $Tpc$ back from stack: $Tpc$ =
    pop();
end
```

6.3.4 Reducing False Positives by Save/Restore

The main issue of a naïve DIFT system is false positives, which we aimed to address using the tunable propagation counter (Section 6.3.1) and analysis and instrumentation from static analysis (Section 6.3.2). Both of the techniques are simply addressing the false positives stemming from within an application. However, once false positives have found their way into the operating system, the false positives grow very quickly across different operating system modules (as we will shown in Section 6.4) and would render the DIFT system unusable due to the sheer amount of false positives.

We explore an idea that further reduces the amount of the false positives in the entire system by leveraging system characteristics. We propose saving and restoring the register file tags (Figure 6.3) whenever the processor changes its privilege level. In other words, if the execution of the userspace program is interrupted, we propose that before the control is transferred to the operating system for asynchronous interrupts, the DIFT hardware automatically saves the register file tags and then zeroes out the current register file tags, so no unnecessary tagged values are accidentally propagated to the operating system. Before the user space program resumes execution, the DIFT hardware would restore the register file tags. Note that the register file tags is only saved and restored when the userspace program is interrupted *asynchronously.* In other words, for *synchronous* privilege level changes such as system calls, the hardware save/restore of the register file tags is not triggered. This allows the DIFT system to continue tracking information across privilege level boundaries to avoid false negatives.

6.4 Implementation and Evaluation

We describe our prototype implementations of the four proposed mitigation techniques and evaluate their overhead and performance in terms of the false positives and false negatives. As described in the Introduction, too many false positives can result in less security, since the DIFT protection system may be turned off temporarily to get real work done – and then one may forget to turn it back on. Of course, allowing false negatives implies the system is not always preventing leaks. Hence, reducing false positives can translate to real-world practicality and usability while minimizing false negatives is necessary for security.

Our baseline DIFT architecture is implemented on the OpenSPARC open source processor [90] simulator. We have modified the stock OpenSPARC T1 processor to track all explicit information flows as described in Section 6.1.1. Table 6.1 lists the propagation rules for the baseline DIFT architecture as well as the tunable propagation employed in our implementation.

We wanted a program that has a pre-existing implicit information flow situation that we could use to show how well our mitigation techniques work. Also the program needs to be small for efficiency, yet realistic for our simulator. We wanted a program that goes through both userspace and the operating system. We found a program that calculates the digits of π, and uses conditional branches to determine how to format the printing of these digits. If we mark the initial values of π as confidential data, then the π program has an implicit information flow based on a sensitive condition used in a conditional branch instruction, and does go from userspace to OS space and back. A simplified version of the program [1] is shown in Figure 6.8.

This example π-calculation program starts with an array `src[]` with 337 elements, all of which we tag as sensitive variables. The tags propagate to variable `q`, which is a function `F()` of `src[]` and other untagged data. These then propagate to other calculations, e.g., `H()`, which is shown in Figure 6.9. Note that the function `G()` does not contain a sensitive condition, since both `k` and `j` are non sensitive. At the end

Table 6.1: Tag propagation rules. Bold fonts indicate rules specific for the tunable propagation. The capital T denotes the tags. The rules in parentheses are implemented depending on the specific instruction that either operates differently according to the condition code register, affects the condition code register or uses the y register. ccr refers to the condition code register in the SPARC architecture. Ty is the tag for the y register, $Tccr$ denotes the tag for the condition code register, and likewise for the source registers ($src1$ or $src2$) and the destination register, $dest$. For load and store instructions, $T[src]$ refers to the tag of the memory location pointed by src, and likewise for $T[dest]$. The memory indexing discussed in Section 6.1.2 are shown in parentheses for the load and store instructions and it depends on the implementation to turn on/off for the tracking of memory indexing.

Instruction Category	**Type**	**Propagation Rule**
Integer Arithmetic	$rs1, rs2$	$Tdest \leftarrow Tsrc1 \cup Tsrc2 \cup \boldsymbol{Tpc}\,(\cup Tccr \cup Ty)$ $Tdest \leftarrow \boldsymbol{Tpc}\,(\cup Tccr)$ $Tdest \leftarrow Tsrc1 \cup \boldsymbol{Tpc}\,(\cup Tccr \cup Ty)$ $Tdest \leftarrow Tsrc2 \cup \boldsymbol{Tpc}\,(\cup Tccr)$
	src, imm	$(Tccr \leftarrow Tsrc1 \cup \boldsymbol{Tpc}\,(\cup Tccr \cup Ty))$ $(Tccr \leftarrow Tsrc1 \cup Tsrc2 \cup \boldsymbol{Tpc}\,(\cup Tccr \cup Ty))$ $(Ty \leftarrow Tsrc1 \cup \boldsymbol{Tpc})$
	Multiply, Divide	$(Ty \leftarrow Tsrc1 \cup Tsrc2 \cup \boldsymbol{Tpc})$
Memory access	Load	$Tdest \leftarrow (Tsrc1\cup)T[src1] \cup \boldsymbol{Tpc}$
	Store	$T[dest] \leftarrow Tsrc1 \cup (Tdest\cup)\boldsymbol{Tpc}$
Branch	Conditional	$\boldsymbol{if(Tccr \neq 0)\{Tpc = Tccr;\ Tpc\_CNT = FLOW\_MAX;\ \}}$ $\boldsymbol{if(Tsrc1 \neq 0)\{Tpc = Tsrc1;\ Tpc\_CNT = FLOW\_MAX;\ \}}$ $\boldsymbol{if(Tsrc1 \cup Tsrc2 \neq 0)\{Tpc = Tsrc1 \cup Tsrc2;\ Tpc\_CNT = FLOW\_MAX;\ \}}$
	Unconditional	None
Every instruction		$\boldsymbol{if(Tpc\_CNT > 0)\{Tpc\_CNT--;\ \}\ elseif(Tpc\_CNT == 0)\{Tpc = 0;\ \}}$

```
1: src[337], k=4000, p, q, t=1000, n=0, dst[1100];
2: for(;src[j=q=0]+=2,--k;)
3:   for(p=1+2*k;j<337;q=F(src[j], k, q, p, t),src[j++]=q/p)
4:     if(G(k, j))
5:       n+=sprintf(&dst[n], "%.3d", taint=H(src[j-2], t, q, p))
6: printf("%s", &dst[0]);
```

Figure 6.8: The simplified version of the π program, to illustrate the effect of implicit information flows.

```
G(k, j) = k != j > 2;
H(a, t, q, p) = a % t
                + q/p/t;
```

Figure 6.9: The `G()` and `H()` functions for the π program.

```
sprintf(buffer, format, number){
    ⋮
  while (number != 0)
  {
    digit = number % 10;
    number /= 10;
      switch (digit)
      {
        case 0: character = '0'; break;
        case 1: character = '1'; break;
           ⋮
      }
      buffer[cnt++] = character;
  }
    ⋮
}
```

Figure 6.10: The core character conversion function that exhibits implicit information flow due to control dependency in `sprintf()`.

of the calculations, the array `dst[]` is written with ASCII characters corresponding to the calculations derived from `src[]`. One of the character conversion functions contains a conditional branch that depends on a tagged value, as shown by the `number` variable in Figure 6.10. In the π example, the tagged value to be used as the `number` in `sprintf()` is the `taint` variable in line 5 of Figure 6.8.

We will show that there is an inadvertent implicit information flow in this little program, hidden in the `sprintf()` library call. We first describe the working of this program. The seemingly complex program can be interpreted as two nested loops and an `sprintf()` function to print ASCII characters to the `dst[]` array. Looking at the program more closely, we can deduce that at least the following variables or memory locations should ideally be tagged as sensitive when the program finishes execution: (1) the entire array `src[]`, the original tag source, (2) 64-bit variable `q`, since `q` is assigned to a value that depends on the values in `src[]` (line 3), (3) the first 1001 bytes (1001 / 8 $\approx$ 126 64-bit words) of `dst[]`, since they get assigned based on the variable `taint` which is based on some function of both the tagged `src[]` and tagged `q` (line 5), and (4) the 64-bit variable `n`, since it depends on the actual number of characters printed by `sprintf()` which depends on the value of `taint` (line 5). `n` is equal to 1001 at the end of the execution.

The core of the `sprintf()` function is a while loop that extracts the remainder of the division of the integer argument (`number` in Figure 6.10) by 10. The while loop keeps executing as long as the quotient is greater than zero, and the quotient is used as the dividend in the next loop iteration. The extracted remainders are then used to lookup the corresponding characters to print. We show the disassembled SPARC assembly instructions for the core of the `sprintf()` function in Figure 6.11.

Looking more closely at the assembly instructions in Figure 6.11 shows that the compare instructions (`cmp`) at lines 22 and 26, sets the tag of the condition code register ($Tccr$), since register `%g2` contains sensitive data `digit`. The subsequent branch-if-equal instruction `be` propagates the tag from the condition code register to the tag of the program counter (Tpc). Therefore the instructions executed after the branches (line 31-34, and 35-38, respectively) will carry the tag value of `digit`. If this implicit information flow is not tracked by the DIFT system, the store-byte instruction (`stb`) at lines 32 and 36 will not be tagged, resulting in false negatives. The corresponding control flow graph of the assembly code of Figure 6.11 is shown in Figure 6.12, with each block showing their line numbers in Figure 6.11. Note that the implicit information flow from the outer `while` loop actually does not matter since there is explicit information flow from `number` to `digit`.

Note particularly that in this program, the `dst[]` array will *only* get tagged through implicit information flow since `sprintf()` prints out the decimal characters based on comparisons of the integer arguments. In summary, a perfect DIFT system would tag a total of (337 + 1 + 126 + 1 =) 465 64-bit words, including the original tag source `src[]`. The actual number may be a little higher due to some temporary variables used in the library function `sprintf()`.

To establish bounds for comparison with our mitigation techniques, we first look at how the baseline DIFT system which is not tracking control dependency, and the naïve DIFT system which tags everything after the tagged branch instruction perform, using this π-calculation example program. Before the start of every experiment, we clear all the leftover tags in the system, and we initialize the tags for the sensitive data (`src[]` in the π-calculation program). We let the program run until completion and then take a snapshot of the system to count the number of memory locations that are tagged. The granularity of the tags is one tag per 64-bit word. In other words, we consider the memory location as tagged even if only one byte within the 64-bit word is tagged.

Basically the baseline DIFT system and the naïve DIFT system set up the lower and upper bounds for the false positives and false negatives performance of our mitigation techniques. The second and third rows of Table 6.2 summarize the results for the baseline and naïve DIFT system. The baseline DIFT system does not track control dependencies, resulting in the entire `dst[]` array being untagged; whereas the naïve DIFT system, although exhibiting no false-negatives, produces more than 6X

```
 3:    104f8:   10 80 00 28     b  10598
 4:    104fc:   01 00 00 00     nop
 5:    10500:   c6 07 bf e4     ld   number, %g3
 6:    10504:   85 38 e0 1f     sra  %g3, 0x1f, %g2
 7:    10508:   81 80 a0 00     mov  %g2, %y
 8:    10518:   82 78 e0 0a     sdiv %g3, 0xa, %g1
 9:    1051c:   82 00 40 01     add  %g1, %g1, %g1
10:    10520:   85 28 60 02     sll  %g1, 2, %g2
11:    10524:   82 00 40 02     add  %g1, %g2, %g1
12:    10528:   82 20 c0 01     sub  %g3, %g1, %g1
13:    1052c:   c2 27 bf e8     st   %g1, digit     ; explicit information flow
                                                    ; from number to digit
14:    10530:   c2 07 bf e4     ld   number, %g1
15:    10534:   85 38 60 1f     sra  %g1, 0x1f, %g2
16:    10538:   81 80 a0 00     mov  %g2, %y
17:    10548:   82 78 60 0a     sdiv %g1, 0xa, %g1
18:    1054c:   c2 27 bf e4     st   %g1, number
19:    10550:   c2 07 bf e8     ld   digit, %g1
20:    10554:   c2 27 bf dc     st   %g1, digit
21:    10558:   c4 07 bf dc     ld   digit, %g2
22:    1055c:   80 a0 a0 00     cmp  %g2, 0         ; test if digit = 0
23:    10560:   02 80 00 08     be   10580          ; tagged branch
24:    10564:   01 00 00 00     nop
25:    10568:   c2 07 bf dc     ld   digit, %g1
26:    1056c:   80 a0 60 01     cmp  %g1, 1         ; test if digit = 1
27:    10570:   02 80 00 08     be   10590          ; tagged branch
28:    10574:   01 00 00 00     nop
                 . . . . . .                        ; test if digit = 2, 3, etc.
29:    10578:   10 80 00 08     b    10598
30:    1057c:   01 00 00 00     nop
31:    10580:   82 10 20 30     mov  0x30, %g1
32:    10584:   c2 2f bf ef     stb  %g1, character ; character = '0'
33:    10588:   10 80 00 04     b    10598
34:    1058c:   01 00 00 00     nop
35:    10590:   82 10 20 31     mov  0x31, %g1
36:    10594:   c2 2f bf ef     stb  %g1, character ; character = '1'
37:    10588:   10 80 00 04     b    10598
38:    1058c:   01 00 00 00     nop
                 . . . . . .                        ; character = '2', '3', etc.
39:    10598:   c2 07 bf e4     ld   number, %g1
40:    1059c:   80 a0 60 00     cmp  %g1, 0         ; check end of while loop
41:    105a0:   12 bf ff d8     bne  10500          ; tagged branch
```

Figure 6.11: The disassembled SPARC assembly instructions for the core function of `sprintf()` in Figure 6.10. Line 39-41 checks for the condition in the outer `while` loop. Line 5-18 calculate `number` and `digit`. Line 21-28 test `digit` for the sensitive `switch` statement and line 31-38 assign the value for `character`.

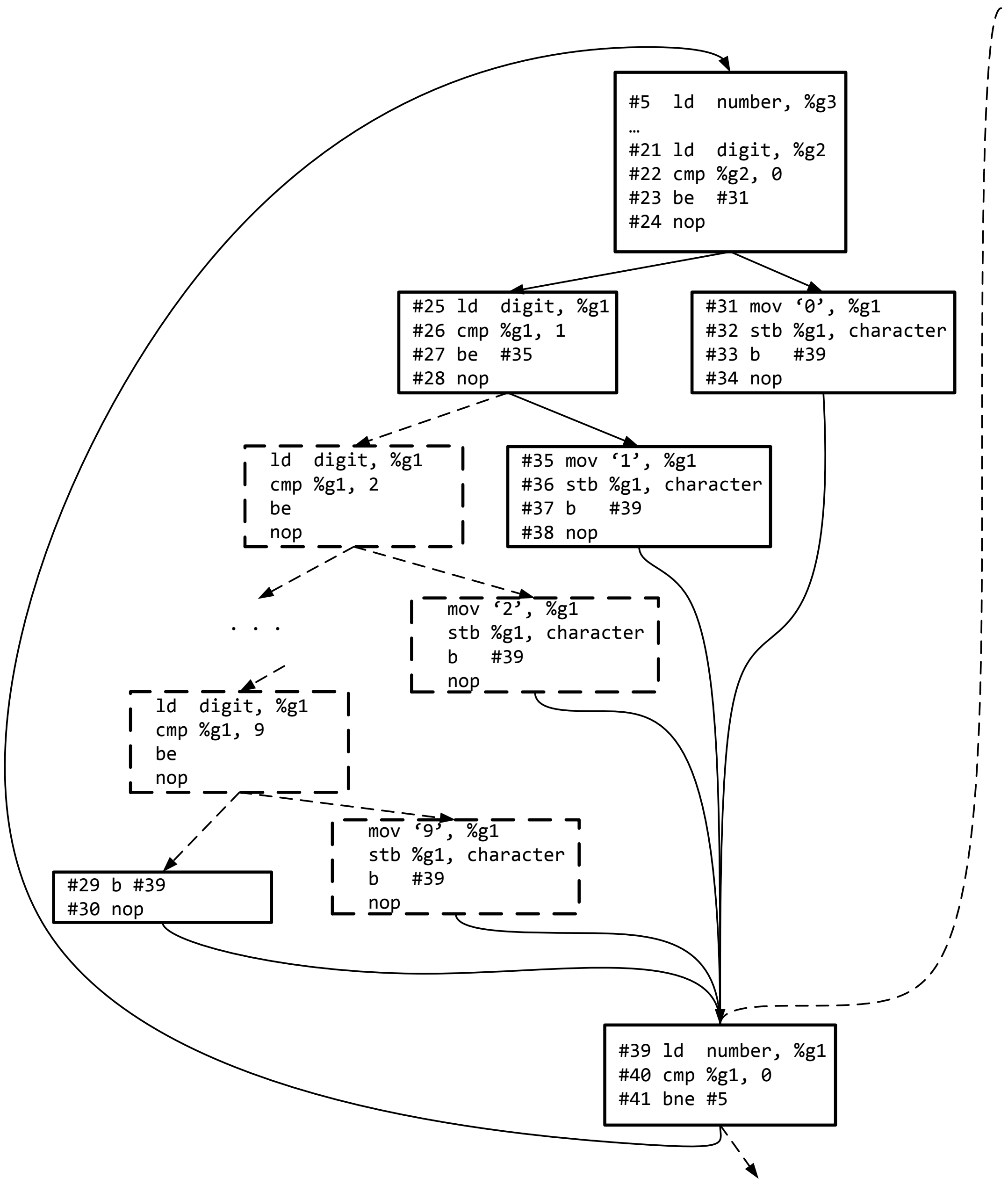


Figure 6.12: The control flow graph of the core function of `sprintf()`. Each block has its corresponding line numbers from Figure 6.11. Note that in this particular example, an ideal $FLOW\_MAX$ value would be 3, to cover the sequence of `nop`, `move` and `stb` instructions after the tagged branches (`be`) in line 23 and 27.

Table 6.2: False positive and false negative performances of DIFT systems with implicit informal flow support. Lines 2 and 3 show the lower and upper bounds given by the baseline DIFT (no tracking for control dependency) and the naïve DIFT system (constant tracking for control dependency), for running the π-calculation program. Each experiment is repeated 10 times and the average values are reported. We pick the zero-false-negative and balanced values from the $FLOW\_MAX$ technique when $FLOW\_MAX$ is equal to 4 and 3, respectively. (see Figure 6.14) and the register file tags save/restore technique when $FLOW\_MAX$ is equal to 5 (see Figure 6.17).

	Tagged words	FP	FN	Notes
Optimal	465	0	0	
Baseline DIFT	339.1	2.1	126	
Naïve DIFT	3154.4	2691.4	0	
$FLOW\_MAX$ w/ zero false negatives	1720.6	1248.6	0	See Fig. 6.14
$FLOW\_MAX$ balanced	520.3	100.2	44.2	See Fig. 6.14
$FLOW\_MAX$ w/ Static binary analysis	485	20	0	
$FLOW\_MAX$ w/ Save/Restore optimal	473.3	8.3	0	See Fig. 6.17

the number of tagged words when comparing to the ideal or perfect DIFT system with 465 tagged words, shown in the first row of Table 6.2. The next four rows show the performance of the mitigation techniques we propose, which we discuss below.

6.4.1 Tunable Propagation

We first describe the implementation and cost for the tunable propagation for control dependency. As pointed out at the end of Section 6.3.1, this mitigation technique requires one additional count-down counter and two additional registers to store the values of $FLOW\_MAX$ and Tpc. Figure 6.13 depicts the additional components needed for this mitigation technique and its connections to the baseline DIFT system.

Figure 6.14 shows the false positives and false negatives analysis of running the π-calculation program using the tunable $FLOW\_MAX$ technique. The columns represent the number of tagged (non-zero tag value) words when $FLOW\_MAX$ is set to the values on the x-axis. The dashed line in Figure 6.14 represents the expected number of tagged 64-bit words in the system that would be achieved by an ideal DIFT system, which is 465 as described previously. In other words, we can expect false-negatives if the total tagged words in the system are below this number and false-positives otherwise.

The bars for the total tagged words in Figure 6.14 are further broken down into different categories that indicate which level of the software hierarchy is responsible for writing to the tagged memory location. For example, "OS" means that the operating

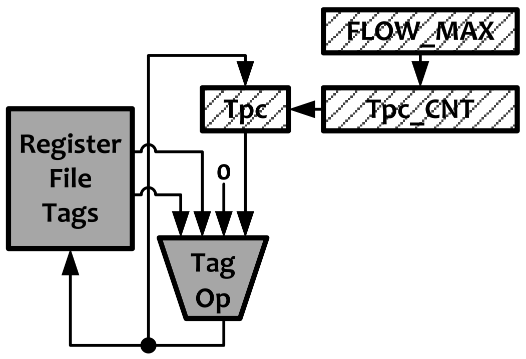


Figure 6.13: The additional components (in stripes) for the tunable propagation of the baseline DIFT system.

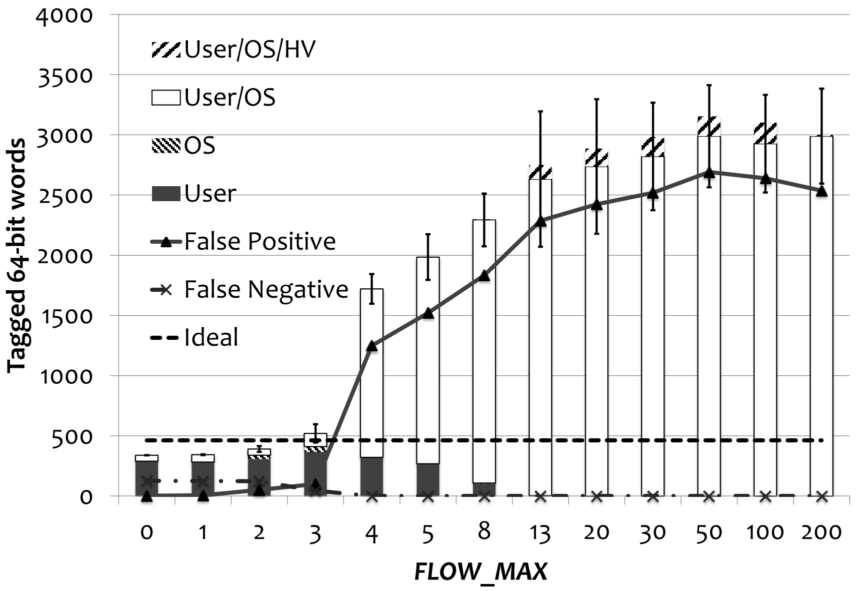


Figure 6.14: False positives and false negatives analysis after running the π-calculation program. The contributions from the hypervisor (HV), User/HV and OS/HV are all zeroes and not plotted on the graph. The x-axis denotes different settings of the $FLOW\_MAX$ values and the y-axis shows the number of tagged 64-bit words in the system. Note that the x-axis is not in linear scale.

system wrote to that memory location with tagged data and “User/OS” means that both the userspace programs and the operating system have written tagged data to the same memory location.

Figure 6.14 shows that for this particular program, the setting of $FLOW\_MAX =$ 3 gives the good balance between false-positives (20%) and false negatives (9%), if the system permits leakage of information below a certain threshold. When $FLOW\_MAX$ is set to 4 or greater, there are no more false-negatives, but false-positives begin to rise sharply (5th and 4th rows of Table 6.2), and continues to increase until it stabilizes at around 3000 tagged words. Note that most of the increase in false positives is for words touched by both the user and the OS, indicating more reductions of false-positives can be achieved by analyzing the OS-contributing factor and employing OS-specific tag clearing techniques, which we will discuss in more detail in Section 6.4.3. At large $FLOW\_MAX$ values, the false-positives is about six times more than the ideal value, which would render the system impractical, since users would most likely turn off any security feature that annoys them too often or simply click *yes* whenever there is a security warning [68]. Therefore, it is very important to determine the value for $FLOW\_MAX$ to ensure maximum security (no false negatives) with minimal interruptions (small false positives). Depending on the security and usability requirements of the system, the value of $FLOW\_MAX$ can be correspondingly adjusted.

6.4.2 Static Binary Analysis and Instrumentation

This section covers both of the Sections 6.3.2 and 6.3.3, since both techniques leverage the information from static binary analysis to instrument the program, although one focuses on reducing false-positives and the other on eliminating false-negatives. Our binary analysis is implemented on the BitBlaze [87] platform.

Applying Algorithm 4 to the π-calculation program and running the program on the same hardware DIFT infrastructure gives the total tagged 64-bit words of 485 which includes 20 false-positives and no false-negatives, compared to the ideal value of 465. This shows that with the assistance of static binary analysis, we can achieve a practical information flow tracking system with no false-negatives and a low false-positive percentage of 4.3%, for this particular program (Table 6.2).

Although no source code is needed since the analysis is done on binaries, this technique requires doing a pre-execution static analysis and adding instructions to the executable binary code. The advantage though is that there need be no guessing of the value of $FLOW\_MAX$, since it is set to cover the scope of the sensitive conditional, and the variables on the untaken path are also tagged sensitive when necessary. Covering the scope is necessary for "soundness" since the application program could be interrupted for an arbitrary number of instructions, and $FLOW\_MAX$ could count down to zero executing OS instructions, before execution returns to the application program. Without this immediate post-dominator scoping, there could be false negatives. The same scoping technique also prevents unnecessary false positives when guessing to set the $FLOW\_MAX$ to a large number for high security – since the clearing of Tpc and setting $FLOW\_MAX$ back to zero at the immediate post-dominator, also reduces unnecessary propagation beyond the scope of the tagged conditional. Covering the untaken path is necessary to ensure no false negatives for dynamic implicit information flow tracking, where only one path of a conditional branch can be taken during execution.

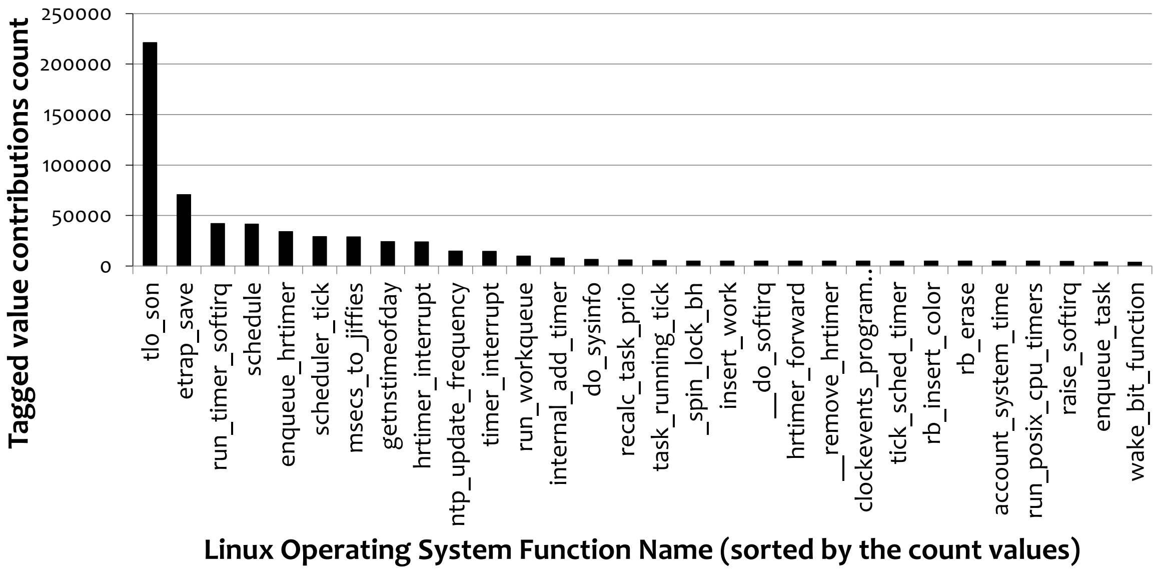


Figure 6.15: The Linux operating system functions that contribute to the tagged values in the system. Each write of a tagged 64-bit word to the memory by the operating system is counted as one contribution. The list is sorted by the amount of contribution from the various operating system functions.

6.4.3 Tag Register Save/Restore

If we analyze the trend given in Figure 6.14, we see that as the $FLOW\_MAX$ value increases, the amount of tagged value contribution from the operating system significantly increases as well. This is due to the fact that once false positives have entered the memory space of the operating system, they propagate further among other operating system components as well. We took one of the runs of our experiment and instrumented the simulation to keep a record of the program counters (PC) within the operating system that contributed to the tagged values and cross referenced the PC addresses with the operating system functions to figure out which modules of the operating system are the major contributors and which modules are the "entry points" of these tagged values into the rest of the operating system. Our analysis is done on the Ubuntu 7.10 with a Linux kernel version of 2.6.22-15-sparc64.

Figure 6.15 shows the distribution of the functions within the operating system that actually contribute to the total tagged values in the system and we find that the top 10 contributing functions within the OS cover more than 75% of the total tagged values within the system. We then look further into the details of these top 10 functions, as listed in Table 6.3 with their contribution percentage and a brief description of their functions within the operating system. From the table we gather that tagged values escape from the userspace into the OS space through three main avenues: (1) register spill/fill routines, (2) trap save routine for preparing the entry into the kernel and (3) timer interrupt routines.

Table 6.3: Brief descriptions of the top 10 operating system functions with the highest tagged value contributions, along with their corresponding frequency of occurrences represented as the percentage of the total aggregate count in Figure 6.15.

Function name	%	Description
`tl0_s0n`	31.9	Register spills.
`etrap_save`	10.2	Preparing for entry into the kernel.
`run_timer_softirq`	6.1	This function runs timers and the timer-tq in bottom half context.
`schedule`	6.0	The main scheduler function.
`enqueue_hrtimer`	4.9	Internal function to (re)start a timer.
`scheduler_tick`	4.2	This function gets called by the timer code.
`msecs_to_jiffies`	4.2	Conversion of time units.
`getnstimeofday`	3.5	Returns the time of day in a timespec.
`hrtimer_interrupt`	3.5	High resolution timer interrupt.
`ntp_update_frequency`	2.2	Updates ticks within the system.
Sum	78.9	

These three main avenues belong to the asynchronous privilege level change category, since they are mainly caused by program side-effects, e.g., page faults or limited amount of hardware register windows[8]. We implement the register file tags save/restore to save the values of the register file tags and zero the tags before the control is transferred to these routines. Subsequently when these routines finish execution and before the control is transferred back to the interrupted application, the hardware save/restore mechanism then restores the register file tags for the particular application, in a similar fashion to a context switch.

Figure 6.16 depicts the additional components needed to perform register file tags save/restore on top of the baseline DIFT system. We use the program counter (pc) to look up the Save/Restore table that points to the memory location pre-assigned to save the register file tags. The value of pc or pc+4 is used depending on whether or not the interrupted instruction is re-executed or not. Likewise, the table is looked up by the hardware to find the saved register file tags when the values are restored upon resuming the application program.

To show the effectiveness of reducing the amount of false positives in the system by the register file tags save/restore mitigation technique, we combine the tunable

[8]Register spill/fill can be caused synchronously (intentionally) by applications, e.g., through recursive or nested function calls. However, it is not deterministic when spill/fill occurs and it varies for different processor architectures. The trap save routines exclude the ones triggered by the application, e.g., system call traps.

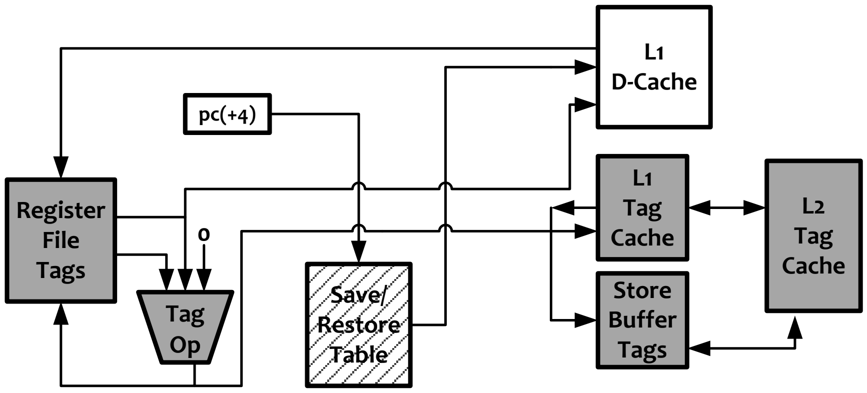


Figure 6.16: Additional DIFT components for register file tags save/restore (in stripes) to address tag pollution in the OS space.

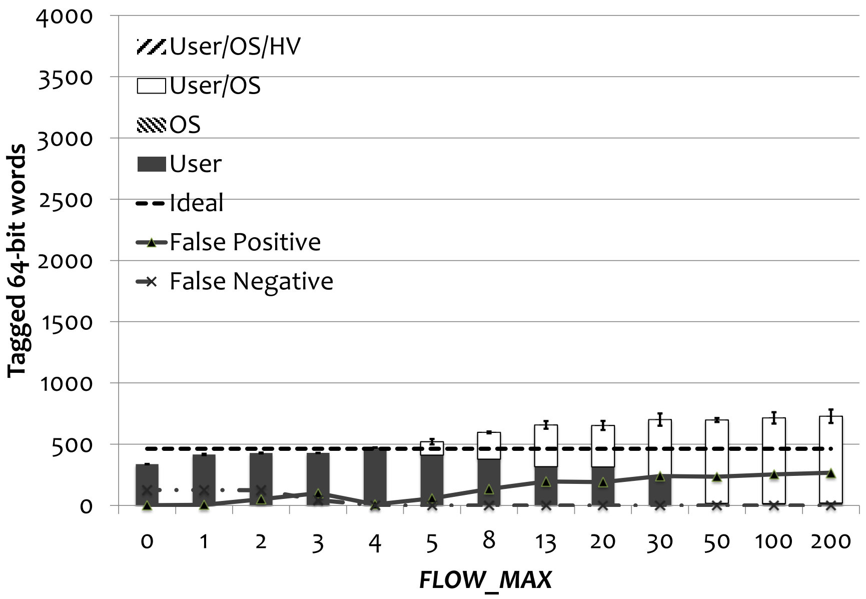


Figure 6.17: False positives and false negatives analysis after running the π-calculation program, with the register file tags save/restore mechanism. Note again that the x-axis is not in linear scale.

propagation mechanism (Section 6.3.1) with the save/restore mechanism and run the same π-calculation program. Figure 6.17 shows the same statistics collected as shown before in Figure 6.14. We can see the reduction of total tagged words by nearly 4X, when compared to the naïve DIFT (tunable propagation) solution (Table 6.2). This greatly increases the practicality and usability of the simple tunable propagation technique, without requiring any static analysis, and instrumentation additions to the application binary.

6.5 Discussion

We first discuss the relation of our implicit information flow solutions with the DataSafe architecture (Chapter 5), using the π example as an illustration, and then we compare the some interesting characteristics between static and dynamic IFT systems.

6.5.1 DIFT and DataSafe Architecture

Tracing the assembly instructions (Figure 6.11) and the control flow graph (Figure 6.12) of the core function of `sprintf()` that exhibits implicit information flow shows that our proposed mechanisms in Chapter 6 is able to track these implicit information flow correctly, without any false-negatives. We now complete the picture by showing how the DataSafe architecture can prevent the tagged data from being output by the application.

The `sprintf()` function on line 5 in Figure 6.8 writes the tagged bytes to the destination array (`dst[]`), which is ultimately written to the screen by the `printf()` function on line 6. The `printf()` function recognizes that it is writing a string of bytes, and calls the `write(1, dst, len)` system call, where `len` denotes the length of the string. In our π program example, the length of the string is 1001, the number of digits of the calculation. Since this system call is a synchronous system call, the save/restore mechanism will not be triggered and the tag of the destination buffer `dst` will propagate into the operating system space. In the operating system, the `write` system call copies the tagged data in the `dst` buffer and writes them to the memory location that is mapped to the frame buffer of the display device. The DataSafe hardware's output control circuitry (Section 5.2.6) will check the tag value of the data to see if the data is allowed to be copied to the frame buffer of the display (mapped in memory range `0x1F10000000` to `0x1F10000050` in our 64-bit Legion simulation infrastructure, see Section 5.3.2) and disallow the data copy if the policy prohibits it. Note also that our implicit information flow mechanisms enable tag propagation across applications, library and the operating system (the π-program, `sprintf()` library function, and the `write()` system call), thus confirming that DataSafe's hardware tracking transcends program and system boundaries.

6.5.2 Static vs. Dynamic

In this chapter, we have thus far focused on the *dynamic* aspect of information flow tracking, and also on adopting and adapting the benefits of *static* IFT techniques dynamically. However, there is one remaining piece that is worth mentioning as interesting future research, and we dedicate this section to discussing some details of it.

Although from our discussion of prior work in IFT techniques in Section 6.2, it seems that static techniques have the positive edge against DIFT since DIFT in theory cannot deal with implicit information flows without being overly restrictive. However, pure static techniques suffer from a major disadvantage – static policy, in addition to requiring access to the source code. RIFLE [94] made a similar observation that static techniques allow the *programmer* to specify the policy, whereas in real life we would like to let the *users* of the programs and the *owners* of the data to specify their own policy.

Another issue with static techniques is static labeling. Since the static techniques, such as the language-based information flow tracking [81], perform type-checking (or label-checking) on the static program source code, the labels to be checked must be known before the program is actually run. However, in real world situations, each instance of the same program may have to deal with data of different security levels. Zheng and Myers [103] introduced the concept of dynamic security labels into static information flow tracking to allow the security labels to be assigned or changed dynamically, thereby allowing for a more realistic usage scenario. They designed a new language that incorporates the concept of dynamic labels and allows the programmers to test for these dynamic labels before performing security-sensitive operations, and they implement the ideas in the Jif language [8]. Jif extends the Java language with static information flow control, supporting this dynamic labeling mechanism.

Below is an example program from [103] in Jif language that demonstrates how a program in Jif controls information flow:

```
1: final label{L} x;
2: Channel{*x} c;
3: int{H} y;
4: switch label(y) {
5:     case (int{*x} z) c.send(z);
6:     else throw new UnsafeTransfer();
7: }
```

The `final` keyword prevents assignments from changing the meaning of types and the variable `x` is declared as of type `label`, to be used as a label for other values. `x` in this example has a security level low (`L`) in a Multi-Level Security (MLS) system. `*x` dereferences the label value of `x`; therefore the channel `c` has the same security level as `x`. `switch label` is a statement in Jif that does the dynamic label testing. In this example, the case will not be executed since the label value of `x` is at not as restrictive as the label value of `y`, which is assigned high (`H`) at line 3. If otherwise, the value of `y` is assigned to `z` and the channel can send out the value. Having understood how dynamic labeling is done in the static language-based information flow tracking, we discuss the variations of IFT schemes and look at the research opportunities.

The technique we presented in Section 6.3.2 and 6.3.3 demonstrate a synergistic approach between static and dynamic IFT analysis – using the information obtained from static analysis to transform the program into a dynamically-enforceable program. However, once a piece of data is tagged or labeled, there is no mechanism to change the tag dynamically. Therefore, it is of future research value to investigate the interactions between the dynamic labeling schemes [103] and the hardware tagging mechanism. For example, if the hardware can dynamically reconfigure to interpret the same tag value differently to enforce a different policy, a piece of sensitive data can be kept open under different user sessions, reducing the setup and termination overhead.

6.6 Summary

In this chapter, we reviewed and categorized the various kinds of information flows within a computer system, whether explicit or implicit. We briefly discussed the advantages and disadvantages of static and dynamic information flow tracking systems and showed how to practically adopt the static techniques dynamically, to achieve an information tracking system that is both secure against implicit information flow due to conditional execution, yet incurs minimal runtime performance overhead. We proposed three simple yet effective techniques to address both the false-positives and false-negatives performances of a DIFT system, i.e., tunable propagation counter, static binary analysis and register file tags save/restore mechanisms. The chapter ends with a discussion of future research ideas to make the hardware tagging technique more flexible in terms of tag representation and dynamic tag values.

Table 6.4 summarizes the comparison of our information flow tracking techniques to various prior work discussed throughout the chapter. The first three columns look at whether or not the techniques require access to the source code and whether or not they address the implicit information flow problem arising from control dependence and the untaken paths in a dynamic program execution, which we address in this chapter. These are important features in today's computing environment. In the next four columns we examine other forms of implicit information flows, i.e., pointer indirection, timing, termination and other side-channels, which are difficult to solve and provide a rich opportunity of future research. These other types of implicit information flow and side-channel attacks are not in our threat model for this thesis. Finally we distinguish pure software techniques from hardware-based techniques. Note that of the techniques that have no access to source code, there are binary translation techniques and hardware techniques. Our solutions are much faster than those using binary translation like LIFT and DTA++, since it takes about 8-20 instructions to emulate a tag propagation operation per instruction with binary translation. The performance overhead is around 1.5X even with parallel execution of the binary translation [70], whereas we have essentially no execution-time overhead due to parallel hardware tag paths, as shown in Figure 6.3. Compared to solutions requiring hardware changes, we cover implicit information flow, which Raksha does not, and we require much less hardware and software changes than GLIFT, thus being more easily deployable.

	Does not require source code	Control dependence	Untaken path	Pointer indirection	Timing dependence	Termination channel	Other side-channel	Does not require new hardware
Fenton [44]	✗	✓ (special return instruction)	✗	✗	✗	✗	✗	✓
Language-based [81]	✗	✓	✓	✓	✓	✓	✓	✓
HiStar [101]	✗	✗	✗	✗	✓	✓	✓	✓
GLIFT [92]	✗	✓ (predicate architecture)	✓ (predicate architecture)	✓ (memory bounds)	✗	✗	✗	✗
RIFLE [94]	✓	✓	? (unclear from the paper)	✗	✗	(Mentioned but not addressed)	✗	✗
LIFT [77]	✓	✗	✗	✗	✗	✗	✗	✓
Egele et al. [41]	✓	✓ (scope of branch)	✗	✓	✗	✗	✗	✓
Panorama [99]	✓	✗ (except for keyboard input)	✗	✓	✗	✗	✗	✓
DTA++ [55]	✓	✓ (static analysis)	✓ (binary instrumenta-tion)	✗	✗	✗	✗	✓
This thesis	✓	✓ (static analysis)	✓ (binary instrumenta-tion)	✗	✗ (partially mitigated)	✗ (partially mitigated)	✗	✗

Table 6.4: Comparison of various information flow tracking techniques.

Chapter 7

Conclusion and Future Work

This thesis explores the security architectures that focus on the protection of data confidentiality and the data-specific policies, such that it is possible to realize the concept of *self-protecting data*. It is of paramount importance today, especially with the shift of the computing paradigm toward ubiquitous computing, where data can be stored, accessed and processed *anywhere*, *anytime* using virtually *any* application that is available.

In Chapter 4 of this thesis, we leverage the SP security architecture [37, 60], which provides a simple yet flexible software-hardware mechanism for protecting a Trusted Software Module (TSM). This enables us to build applications to express and enforce different security policies, without depending on the operating system which may contain vulnerabilities or be malicious. We demonstrated the implementation of an originator-controlled (ORCON) distributed information sharing policy for documents. Such an access control policy is difficult to achieve with only MAC or DAC mechanisms. Our modified *vi* application is a proof-of-concept of the effectiveness of the SP hardware-software architecture. Furthermore, to enable such data protection policy for arbitrary applications, we developed a general methodology for *trust-partitioning* an application, which is useful not only for our information sharing policy, but more generally for separating out the security-critical parts of an application.

Chapter 5 takes a step further to see how we can achieve similar data protection, *without* needing to partition an application into trusted and untrusted parts, which would have to be done for each application. We want to protect legacy code as well. We indeed achieve this by enabling owners of sensitive data to define a security policy for their encrypted data, then allowing authorized users and *unmodified* third-party applications to decrypt and use this data, with the assurance that the data's confidentiality policy will be enforced. Furthermore, DataSafe architecture prevents illegitimate secondary dissemination of protected data by authorized users by preventing the plaintext data from leaking out of these authorized use sessions. Data is protected even if transformed and obfuscated, across applications and user-system boundaries.

Our DataSafe software-hardware architecture uses enhanced dynamic information flow tracking (DIFT) mechanisms to persistently track and propagate data in-use, and to perform nonbypassable output control to prevent leaking of confidential data. Because this is done in hardware, performance overhead is minimal. However, unlike previous hardware solutions, our software components support flexible security policies that bridge the semantic gap between software flexibility and efficient hardware-enforced policies. In addition, our solution is application-independent, thus supporting both legacy and new but unvetted applications and, more importantly, it provides the separation of data protection from applications, which we believe is the right architectural abstraction.

In Chapter 6, we tackle the problem of implicit information flow. We propose several alternative solutions, which do not require access to the source code. We reduce the false positives significantly while ensuring no false negatives, thus making the system more usable and deployable, without sacrificing security.

7.1 Future Work

There are a number of possible future research directions that can be extended beyond this thesis. We list a few of the short- and long-term research ideas below:

- As we pointed out in Chapter 3, this thesis assumes that trusted paths exist between the user input and the microprocessor, and between the microprocessor and the display. Nevertheless, it is interesting to explore how to incorporate these peripheral devices into the trusted computing base and how to construct the architecture such that the hardware policy checking (Chapter 5) can be delegated to the devices, if the devices also recognize the policy tags of the data. This is of particular interest in light of the rapid development of System-on-Chip (SoC) computing devices where most of the system functionalities are integrated on-chip. If the hardware policy enforcement mechanism can be extended to other components on the same chip, then it is possible to achieve system-level policy enforcement, as opposed to the processor-level enforcement in Chapter 5.
- Once we have the data confidentiality protections proposed in this thesis, the natural next step is to extend the data confidentiality protection across machines. This is different from the current protection where data traveling between machines are encrypted and thus protected. However, we would like to specify conditions where data are safe to travel between computing devices unencrypted, e.g., through the use of remote attestation to verify the authenticity of the connected device before sending out plaintext through secure communication channels.

Bibliography

[1] A Small C Program for Calculating 1000 Digits of π. http://www.boo.net/~jasonp/pigjerr.

[2] Adobe Acrobat Family. http://www.adobe.com/products/acrobat.

[3] Advanced Access Content System (AACS). http://www.aacsla.com/home.

[4] Automatic Key-Finding for AES. https://citp.princeton.edu/memory-content/src/aeskeyfind-1.0.tar.gz.

[5] Automatic Key-Finding for RSA. https://citp.princeton.edu/memory-content/src/rsakeyfind-1.0.tar.gz.

[6] Content Scramble System (CSS). http://www.dvdcca.org/css/.

[7] eCryptfs: Enterprise Cryptographic Filesystem. https://launchpad.net/ecryptfs.

[8] Jif: Java + Information Flow. http://www.cs.cornell.edu/jif/.

[9] Mandatory Access Control of FreeBSD. http://www.freebsd.org/doc/handbook/mac.html.

[10] OpenSSL: The Open Source Toolkit for SSL/TLS. https://www.openssl.org/.

[11] RubyForge: Home for Open Source Ruby Projects. https://rubyforge.org/.

[12] Secure Information Sharing Architecture (SISA) Alliance. http://www.sisaalliance.com/.

[13] Security-Enhanced Linux. http://www.nsa.gov/research/selinux/index.shtml.

[14] The Traditional vi. http://ex-vi.sourceforge.net/.

[15] Xen Security Advisory 7 (CVE-2012-0217) - PV Privilege Escalation, 2012. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0217.

[16] Mohammed I. Al-Saleh and Jedidiah R. Crandall. On Information Flow for intrusion Detection: What If Accurate Full-System Dynamic Information Flow Tracking Was Possible? In *Proceedings of the 19th Workshop on New Security Paradigms*, pages 17–32, 2010.

[17] Ross J. Anderson and Markus G. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 125–136, 1998.

[18] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *Proceedings of 16th USENIX Security Symposium*, pages 15:1–15:16, 2007.

[19] Jan Camenisch and Markus Stadler. Efficient Group Signature Schemes for Large Groups (Extended Abstract). In *Proceedings of the 17th Annual Interna-*

tional Cryptology Conference on Advances in Cryptology, pages 410–424, 1997.

[20] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. Anti-Taint-Analysis: Practical Evasion Techniques Against Information Flow Based Malware Defense. Technical report, Secure Systems Lab at Stony Brook University, 2007.

[21] 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*, pages 143–163, 2008.

[22] David Challener, Kent Yoder, Ryan Catherman, and David Safford. *A Practical Guide to Trusted Computing*, chapter 15, pages 271–276. IBM Press, 2008.

[23] David Champagne. *Scalable Security Architecture for Trusted Software*. Phd thesis, Princeton University, 2010.

[24] David Champagne, Reouven Elbaz, and Ruby B. Lee. The Reduced Address Space (RAS) for Application Memory Authentication. In *Proceedings of the 11th International Conference on Information Security*, pages 47–63, 2008.

[25] David Champagne and Ruby B. Lee. Scalable Architectural Support for Trusted Software. In *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture*, pages 1–12, 2010.

[26] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 13–28, 2003.

[27] David Chaum and Eugène Van Heyst. Group Signatures. In *Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques*, pages 257–265, 1991.

[28] Lidong Chen and Torben P. Pedersen. New Group Signature Schemes. In *Proceedings of the 13th Annual International Conference on Theory and Application of Cryptographic Techniques*, pages 171–181, 1994.

[29] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, 2008.

[30] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A Software-Hardware Architecture for Self-Protecting Data. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[31] Yu-Yuan Chen and Ruby B. Lee. Hardware-Assisted Application-Level Access Control. In *Proceedings of the 12th International Conference on Information Security*, pages 363–378, 2009.

[32] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-Inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *Proceedings of the 20th USENIX Security Symposium*, pages 139–154, 2011.

[33] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime Via Whole System Simulation. In *Proceed-*

ings of the 13th USENIX Security Symposium, pages 321–336, 2004.

[34] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a Flexible Information Flow Architecture for Software Security. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, pages 482–493, 2007.

[35] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th International Conference on Information Security*, pages 346–360, 2011.

[36] Jeffrey S. Dwoskin, Mahadevan Gomathisankaran, Yu-Yuan Chen, and Ruby B. Lee. A Framework for Testing Hardware-Software Security Architectures. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 387–397, 2010.

[37] Jeffrey S. Dwoskin and Ruby B. Lee. Hardware-Rooted Trust for Secure Key Management and Transient Trust. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 389–400, 2007.

[38] Jeffrey S. Dwoskin, Dahai Xu, Jianwei Huang, Mung Chiang, and Ruby B. Lee. Secure Key Management Architecture Against Sensor-Node Fabrication Attacks. In *Proceedings of the 50th Annual IEEE Global Telecommunications Conference*, pages 166–171, 2007.

[39] Petros Efstathopoulos and Eddie Kohler. Manageable Fine-Grained Information Flow. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 301–313, 2008.

[40] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30, 2005.

[41] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic Spyware Analysis. In *Proceeding of the USENIX Annual Technical Conference*, pages 18:1–18:14, 2007.

[42] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B. Lee, Nachiketh Potlapally, and Lionel Torres. Transactions on Computational Science IV. chapter Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines, pages 1–22. 2009.

[43] Jeremy Epstein. Fifteen Years after TX: A Look Back at High Assurance Multi-Level Secure Windowing. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 301–320, 2006.

[44] J. S. Fenton. Memoryless Subsystems. *The Computer Journal*, 17(2):143–147, February 1974.

[45] Bryan Ford and Russ Cox. Vx32: Lightweight User-Level Sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference*, pages 293–306, 2008.

[46] B. Gassend, G.E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proceedings of the 9th IEEE International Symposium on High Performance Computer Architecture*,

pages 295–306, 2003.

[47] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[48] Shafi Goldwasser and Silvio Micali. Probabilistic Encryption & How to Play Mental Poker Keeping Secret All Partial Information. In *Proceedings of the 14th annual ACM Symposium on Theory of Computing*, pages 365–377, 1982.

[49] Richard Graubart. On The Need for A Third Form of Access Control. In *12th National Computer Security Conference Proceedings*, pages 296–303, 1989.

[50] Peter Gutmann. Secure Deletion of Data From Magnetic and Solid-State Memory. In *Proceedings of the 6th Conference on USENIX Security Symposium*, pages 77–89, 1996.

[51] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the 17th Conference on USENIX Security Symposium*, pages 45–60, 2008.

[52] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. http://www.intel.com/Assets/PDF/manual/253665.pdf.

[53] Intel. Isolated Execution. http://software.intel.com/en-us/articles/isolated-execution/.

[54] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[55] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[56] Aggelos Kiayias, Yiannis Tsiounis, and Moti Yung. Group Encryption. In *Proceedings of the 13th International Conference on Theory and Application of Cryptology and Information Security*, pages 181–199. 2007.

[57] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, 1999.

[58] Ulrich Kohl, Jeffrey Lotspiech, and Stefan Nusser. Security for the Digital Library – Protectiong Documents Rather Than Channels. In *Proceedings of the 9th International Workshop on Database and Expert Systems Applications*, page 316.

[59] Abhishek Kumar. Discovering Passwords in the Memory, 2003. White Paper, Paladion Networks.

[60] Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 2–13, 2005.

[61] John Leyden. Blu-ray DRM Defeated: Copy-protection Cracked Again, 2007. http://www.theregister.co.uk/2007/01/23/blu-ray_drm_cracked/.

[62] Zhenkai Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 182–191, 2003.

[63] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the 19th ACM symposium on Operating systems principles*, pages 178–192, 2003.

[64] Enrico Lovat and Alexander Pretschner. Data-Centric Multi-Layer Usage Control Enforcement: A Social Network Example. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, pages 151–152, 2011.

[65] C. J. McCollum, J. R. Messing, and L. Notargiacomo. Beyond the Pale of MAC and DAC – Defining New Forms of Access Control. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 190–200, 1990.

[66] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[67] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 315–328, 2008.

[68] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do Windows Users Follow the Principle of Least Privilege? Investigating User Account Control Practices. In *Proceedings of the 6th Symposium on Usable Privacy and Security*, pages 1–13, 2010.

[69] Microsoft Security Bulletin MS12-042. Vulnerabilities in Windows Kernel Could Allow Elevation of Privilege, 2012. http://technet.microsoft.com/en-us/security/bulletin/MS12-042.

[70] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. Dynamic Information Flow Tracking on Multicores. In *Proceedings of the Workshop on Interaction Between Compilers and Computer Architectures*, 2008.

[71] Kenneth Ocheltree, Steven Millman, David Hobbs, Martin Mcdonnell, Jason Nieh, and Ricardo Baratto. Net2Display: A Proposed VESA Standard for Remoting Displays and I/O Devices over Networks. In *Proceedings of the 2006 Americas Display Engineering and Applications Conference*, 2006.

[72] Oracle. Solaris Operating System Source Code Guidelines. http://goo.gl/Ftwkc.

[73] Jaehong Park and Ravi Sandhu. Towards Usage Control Models: Beyond Traditional Access Control. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, pages 57–64, 2002.

[74] Jaehong Park and Ravi Sandhu. The UCON_{ABC} Usage Control Model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.

[75] Cole Petrochko. DHC: EHR Data Target for Identity Thieves, 2011. http://www.medpagetoday.com/PracticeManagement/InformationTechnology/30074.

[76] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. Mechanisms for Usage Control. In *Proceedings of the 3rd ACM Symposium on Information,*

Computer and Communications Security, pages 240–244, 2008.

[77] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[78] Reuters. Path Fumble Highlights Internet Privacy Concerns. `http://goo.gl/vonKy`.

[79] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–196, 2007.

[80] Joanna Rutkowska. The MS-DOS Security Model. `http://theinvisiblethings.blogspot.com/2010/08/ms-dos-security-model.html`.

[81] A. Sabelfeld and A.C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[82] Arijit Saha and Nilotpal Manna. *Digital Principles and Logic Design*. Jones and Bartlett Publishers, Inc., 1st edition, 2007.

[83] W. Shi, J.B. Fryman, G. Gu, H.-H.S. Lee, Y. Zhang, and J. Yang. InfoShield: A Security Architecture for Protecting Information Usage in Memory. In *Proceedings of the 12th IEEE International Symposium on High Performance Computer Architecture*, pages 222–231, 2006.

[84] Arrvindh Shriraman and Sandhya Dwarkadas. Sentry: Light-Weight Auxiliary Memory Access Control. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 407–418, 2010.

[85] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 161–174, 2006.

[86] Asia Slowinska and Herbert Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 61–74, 2009.

[87] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, Newso James, 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*, pages 1–25, 2008.

[88] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 160–171, 2003.

[89] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution Via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming*

Languages and Operating Systems, pages 85–96, 2004.

[90] Sun Microsystems. OpenSPARC T1 Microarchitecture Specification, 2006.

[91] Alexander Tereshkin. Evil Maid Goes After PGP Whole Disk Encryption. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, page 2, 2010.

[92] 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 *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 109–120, 2009.

[93] Trusted Computing Group. Trusted Platform Module. `https://www.trustedcomputinggroup.org/home`.

[94] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, 2004.

[95] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully Homomorphic Encryption Over the Integers. In *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*, pages 24–43, 2010.

[96] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and Event Processes in the Asbestos Operating System. *ACM Trans. Comput. Syst.*, 25, December 2007.

[97] M.S. Wang and R.B. Lee. Architecture for a Non-Copyable Disk (NCdisk) Using a Secret-Protection (SP) SoC Solution. In *Proceedings of the 41st IEEE Asilomar Conference on Signals, Systems and Computers*, pages 1999–2003, 2007.

[98] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian Memory Protection. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, 2002.

[99] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 116–127, 2007.

[100] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. TightLip: Keeping Applications from Spilling the Beans. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[101] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, 2006.

[102] Xinwen Zhang, Francesco Parisi-Presicce, Ravi Sandhu, and Jaehong Park. Formal Model and Policy Specification of Usage Control. *ACM Trans. Inf. Syst.*

Secur., 8(4):351–387, 2005.

[103] Lantian Zheng and Andrew C. Myers. Dynamic Security Labels and Static Information Flow Control. *Int. J. Inf. Secur.*, 6:67–84, March 2007.

Appendix A

List of File Access Application Programming Interfaces (APIs) in C and Ruby Language

A.1 File Access APIs in C

	Function	**Description**
File Access	`fopen`	Opens a file
	`freopen`	Opens a different file with an existing stream
	`fflush`	Synchronizes an output stream with the actual file
	`fclose`	Closes a file
	`setbuf`	Sets the buffer for a file stream
	`setvbuf`	Sets the buffer and its size for a file stream
Direct I/O	`fread`	Reads from a file
	`fwrite`	Writes to a file
Unformatted I/O	`fgetc`	Reads a byte from a file stream
	`fgets`	Reads a byte string from a file stream
	`fputc`	Writes a byte to a file stream
	`fputs`	Writes a byte string to a file stream
	`ungetc`	Puts a byte back into a file stream
Formatted	`fscanf`	Reads formatted byte input from a file stream
	`vfscanf`	Reads formatted input byte from a file stream using variable argument list
	`fprintf`	Prints formatted byte output to a file stream

I/O	`vfprintf` `vsprintf`	Prints formatted byte output to a file stream using variable argument list
File Positioning	`ftell`	Returns the current file position indicator
	`fgetpos`	Gets the file position indicator
	`fseek`	Moves the file position indicator to a specific location in a file
	`fsetpos`	Moves the file position indicator to a specific location in a file
	`rewind`	Moves the file position indicator to the beginning in a file
Error Handling	`feof`	Checks for the end-of-file
	`ferror`	Checks for a file error
Operations on Files	`remove`	Erases a file
	`rename`	Renames a file

We need to redirect File Positioning APIs since the encrypted file may have a different size from the decrypted file. Therefore the redirection mechanism needs to keep track of the file positions of the DataSafe-protected files, similarly for the Error Handling APIs. Other file access APIs need not be redirected for DataSafe-protected file. For example, the `tmpfile` function that returns a pointer to a temporary file needs not be redirected since DataSafe-protected file will not be a temporary file. Overall, out of all 68 File Access APIs in C, 27 of them need to be redirected, as shown in the above table.

A.2 File Access APIs in Ruby

Class	**Method Type**	**Function**	**Description**
`IO`	Public Class Methods	`foreach(){block}`	Executes the block for every line in the file
		`new`	Returns a new `IO` object
		`open`	A synonym for new
		`open(){block}`	Like `open`, except that the `IO` object will automatically be closed when the block terminates
		`read`	Opens the file, optionally seeks to the given offset, then returns length bytes (defaulting to the rest of the file)
		`readlines`	Reads the entire file specified by name as individual lines, and returns those lines in an array

		sysopen	Opens the given path, returning the underlying file descriptor
IO	Public Instance Methods	bytes	Returns an enumerator that gives each byte in a file
		each_char{block}	Calls the given block once for each character in a file, passing the character as an argument
		close	Closes a file and flushes any pending writes to the operating system
		each each_line	Executes the block for every line in a file
		each_byte	Calls the given block once for each byte in a file
		each_char	Calls the given block once for each character in a file
		eof eof?	Returns true if at end of file
		flush	Flushes any buffered data within a file to the underlying operating system
		fsync	Immediately writes all buffered data in a file to disk
		getc	Gets the next 8-bit byte from a file
		gets	Reads the next line from a file
		lineno	Returns the current line number in a file
		lines	Returns an enumerator that gives each line in a file
		pos tell	Returns the current offset (in bytes) of a file
		pos =	Seeks to the given position (in bytes) in a file
		print	Writes the given object(s) to a file
		printf	Formats and writes to a file
		putc	Writes the character to a file
		puts	Writes each element on a new line to a file
		read	Reads at most length bytes from a file
		read_nonblock	Reads at most maxlen bytes from a file using `read(2)` system call after `O_NONBLOCK` is set for the underlying file descriptor

		`readchar`	Reads a character as with `getc`
		`readline`	Reads a line as with `gets`
		`readlines`	Reads all of the lines in a file, and returns them in an array
		`readpartial`	Reads at most maxlen bytes from the file
		`reopen`	Reassociates a stream with the another stream
		`rewind`	Positions to the beginning of a file, resetting `lineno` to zero
		`seek`	Seeks to a given offset in the file
`IO`	Public Instance Methods	`sysread`	Reads integer bytes from a file using a low-level read and returns them as a string
		`sysseek`	Seeks to a given offset in a file
		`syswrite`	Writes the given string to a file using a low-level write
		`ungetc`	Pushes back one character onto a file
		`write`	Write a given string to a file
		`write_nonblock`	Write a given string to a file using `write(2)` system call after `O_NONBLOCK` is set for the underlying file descriptor
`File`	Public Class Methods	`new`	Opens the file and returns a new `File` object
		`rename`	Renames the given file to the new name
		`truncate`	Truncates the file
		`delete` `unlink`	Deletes the files
	Public Instance Methods	`flock`	Locks or unlocks a file
		`truncate`	Truncates the file

Out of all 67 public methods in the `IO` class in Ruby 1.8.7, we need to redirect 44 of them, whereas, we need to redirect 7 out of 61 public methods in the `File` class, as listed in the above table.