# Modularizing Trust: A Framework for Cloud Storage Security

Simon Sibomana
ss2bs@virginia.edu

***Abstract*—Currently, customers of cloud storage providers (CSPs), such as Dropbox and Copy, have to trust the CSP with access to their data. Alternatives, require trusting some other vendor who provides a client-side encryption module. We propose a framework that enables the distribution of trust over the components of the system used to synchronize the user's data to the CSP's infrastructure. Our approach consists of dividing the system into isolated components, with limited capabilities. We rely on core operating system mechanisms to limit the components' capabilities, and provide the user with configuration scripts that are easily auditable. Assuming that multiple malicious components do not collude, we argue that our framework provides strong security guarantees. We have implemented three configurations of our framework, each addressing different security requirements.**

## I. Introduction

Currently, customers of Cloud Storage Providers (CSP), such as Dropbox (https://www.dropbox.com) and Copy (https://www.copy.com), have to trust the CSP with access to their data. Yet for personal, commercial, or legal reasons, the customers may want (or be required) to restrict access to the data to authorized parties. Customers may be storing personal information in the cloud that might cause embarrassment if made public [1], or confidential business data that might compromise their competitiveness. In addition to protecting the confidentiality of their data, CSP customers would not want to lose access to their data [2] or have it modified in an unauthorized manner [3]. Many CSPs encrypt their customer's data with a key, generated and stored by the CSP. This does not protect the data from internal, or external [4]–[6], attacks.

To further ease the confidentiality concerns of customers, several CSPs, including Tresorit (https://tresorit.com/), Mozy (https://mozy.com/), and Wuala (https://www.wuala.com/), provide client-side encryption, where the data is encrypted on the customer's machine, by software provided by the CSP. For users of CSPs that don't provide client-side encryption, there are third party solutions such as Viivo (https://www.viivo.com/), and Boxcryptor (https://www.boxcryptor.com/), that encrypt the user's data before sending it to a CSP.

These client-side encryption solutions (provided by the CSP or a third party) may enhance the security of the user's data, however, they still require trusting the CSP (or third pary), whose software runs on the user's device, with full access to the user's data and network.

We focus on the scenario where the CSP software, running on the user's device, is untrusted. The Cloud Service Provider may be malicious, or its client-side software may have a vulnerability that can be exploited.

The goal of this work is to limit the trust placed on software obtained from external parties. To this end, we present a framework that distributes trust over the components of a system used to sync the user's data to the cloud. Our approach divides the system into isolated components, and limits each component's privileges to only those required to perform its function. We argue that, as long as the operating system and the isolation mechanism are trusted, and none of the independent components collude to compromise the user's security, our framework strongly guarantees the confidentiality of the user's data.

We also consider the scenarios where some, but not all, of the components are malicious and collude (via overt or covert channels), and argue that for most of these collusion scenarios, our framework can be configured to provide strong confidentiality guarantees.

## II. Related Work

Our work focusses on the scenario of a user running software provided by a Cloud Storage Provider, that we assume is untrusted. The goal is to enable users to protect the confidentiality of their data. We summarize prior work on protecting the confidentiality and integrity of the data of users of CSP services, as well as the work on confining, and limiting the privileges of, untrusted software running on the user's device.

### *Cloud Storage Security*

Kamara and Lauter [7] propose a modular design with separate components responsible for ensuring the confidentiality and integrity of the user's data, granting access to the data, and allowing an authorized user to search over the encrypted data. The authors emphasize that the application implementing this design should be open-source, or should be implemented/verified by a party other than the cloud service provider. Danezis and Livshits [8] also propose a design that comprises encrypting and hashing the data locally before sending it to the cloud. However, none of these architectures protect against a malicious CSP client-side software.

Bugiel et al. [9] propose a multi-cloud architecture to protect the confidentiality of the user's data. The proposed solution involves the use of two clouds, one of which is trusted by the user, to encrypt and verify the user data and

operations performed on the second untrusted cloud. The proposed solution requires the user to have enough resources to setup a private cloud, or to trust a third party with the security of their data. Neither option is ideal in a scenario where the user doesn't trust any third party, and might not have the resources to setup their own private cloud.

*Isolating Untrusted Software*

Viswanathan and Neuman [10] present a survey of techniques for isolating processes on computer systems, and include a comprehensive list of systems that use one or more of these techniques. One of the techniques presented, sandbox-based isolation, offers users a lightweight option for confining an untrusted program and controlling its execution. The user specifies policies which restrict the actions of the untrusted program on the user's system.

One of the common techniques for sandboxing software applications is system call interposition, which works by intercepting the system calls of the specified application, and based on policies configured by the user, decides whether to permit or deny the action requested by the application. Systems that employ this technique include systrace [11], and MBOX [12].

Vijayakumar et al. [13] present process firewalls, a mechanism that prevents unauthorized access of resources, by examining the internal state of a process, as well as the resource context to determine whether the requested access could result in unauthorized access of the resource. The main goal of the process firewall is to protect a trusted process from being tricked into leaking sensitive information, however it can also be used to confine an untrusted application.

*Privilege Separation*

Our approach is based on the principle of least privilege [14]. Several works [15], [16], have used the approach of splitting a large untrusted application into smaller components that run with limited privileges, to mitigate against the threat of the application compromising the security of the user's data.

Andrea Bittau et al. [15] present OS primitives to allow a programmer to split an application into isolated default-deny components, as well as a tool for determining the privileges required by each component. The programmer needs to determine and specify which memory objects should be accessed by each component, and the access rights for each of the components. The tool, analyzes the run-time memory access behavior of an application, and summarizes for the programmer which code requires which memory access privileges.

Aaron Blankstein and Michael Freedman [16] use a similar approach, but focus on limiting the types of accesses that can be made by a web application to shared storage such as a database. They present a tool for monitoring the queries issued by the different components of the application, during a trusted learning phase, in order to determine the constraints that should be placed on the queries issued by different components.

Both works assume the application being split, though untrusted, is not malicious, and both require the user to trust the application provided to not be malicious, and to correctly split the application into isolated default-deny components.

## III. Threat Model

Our work considers the scenario where the CSP software application, as well as any encryption software, installed on the user's device, may be malicious or compromised. We therefore don't trust these applications, obtained from external parties, not to compromise the confidentiality of the user's data. We assume that a malicious software application could leak the user's unencrypted data, or cryptographic keys, to some unauthorized party.

We assume that the user's hardware and operating system are not malicious, and operate correctly.

## IV. Security Framework

Our approach is to isolate the untrusted applications from the rest of the user's system, limiting the privileges of these untrusted applications to only those necessary to perform their function, and allowing them to exchange data, with each other, and with the user's system, through narrow interfaces.

One of the main advantages of our framework is its extensibility. Different configurations of the framework can be easily implemented based on the user's threat assumptions, and security objectives.

### A. Design

Our framework consists of two types of modules, CSP client modules, and Encryption modules, isolated from each other, and from the rest of the user's system.

The **isolation mechanism** isolates a module by enforcing the designated access policy for each module. The access policy specifies what components of the user's system the module is allowed to access.

The **CSP client module** contains the CSP client application, and exchanges encrypted data with the Encryption module connected to it. The CSP client application has no access to the rest of the user's system, it can only access the data inside this module. This module is allowed to connect to the Internet, to allow the CSP client to sync the user's data to the cloud.

The **Encryption module** contains an encryption application that encrypts, or decrypts, the received data. Network connectivity is disabled in all of the Encryption modules to prevent the leakage of cryptographic keys, as well as the user's plaintext data, in the case of the inital Encryption Module.

The modules connect to each other, and the user's system, via shared directories that are mounted from the user's file system.

By isolating the CSP client from the rest of the user's system and the encryption environment, and allowing it to only access encrypted user data, our framework obviates the need for a customer to trust the CSP client. Isolating the encryption software, which is also a third-party software (or may be composed from a third-party cryptographic library), and preventing it from having Internet access, is done to

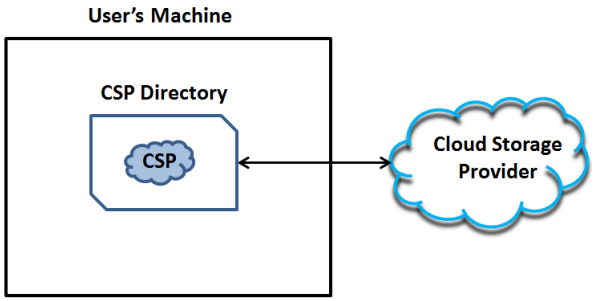


Figure 1. Conventional Configuration

ensure that neither the unencrypted data, nor cryptographic keys are leaked to unauthorized parties. Our framework thus ensures that in order to compromise the data's confidentiality, all, or at least some, of the untrusted parties would need to collude. The extensibility of our framework—the user can select multiple Encryption modules, each running different encryption software, and multiple CSP modules, each with a different CSP client—makes it suitable for achieving many security objectives.

### B. Configurations

To illustrate the versatility of our framework, in terms of the threats it can be used to address, we describe several configurations below.

*1) Conventional:* The Conventional configuration is one where the user simply sets up a CSP client on the machine and synchronizes data to the CSP's infrastructure via the CSP client's directory. In this configuration, the user does not employ our framework.

This configuration assumes that the CSP is trusted, and that the user's OS, hardware, as well as all software applications on the machine, are also trusted. The CSP client can see all data on the host, and send anything over the network.

*2) Split-Trust:* The Split-Trust configuration, consists of a single encryption module and a single CSP client module, connected as shown in Figure 2. The "User" directory, contains the user's unencrypted data. It is mounted in the Encryption module as the "Dec" directory. After the data is encrypted, it is placed in the "Enc" directory, which is synchronized with the CSP client's "CSP" directory.

This configuration addresses the threat that the CSP client application, as well as the encryption software, installed on the user's device, may be malicious or compromised, and may attempt to leak the user's unencrypted data, or cryptographic keys. Though the encryption software may be malicious, we assume that it correctly encrypts the user's data, and that it employs a strong encryption algorithm. We also assume that none of the untrusted software applications, that is, the CSP client and encryption applications, are colluding.

The purpose of this configuration is to protect the confidentiality of the user's data from being compromised by either the CSP software or the encryption software. We have implemented this configuration, and provide a description of the implementation (Section 4.3), as well as our security analysis of the implementation (Section 4.4.1).

*3) Counter-Collusion:* The Split-Trust configuration provides a strong guarantee of the confientiality of the user's data, when none of the untrusted components, the encryption application and the CSP client, collude. In a scenario where these components collude to leak the user's data, the Split-Trust configuration does not guarantee the confidentiality of the data. We therefore designed, and implemented, the Counter-Collusion configuration to address this weakness.

The Counter-Collusion configuration, shown in Figure 3, consists of multiple encryption modules and a single CSP module. The directories, "User" directory and "Dec", contain the data that is to be encrypted, and the "Enc" directories, the encrypted data. The numbers 1 and 2 represent the sequence in which the data is encrypted.

This configuration is designed with the same trust assumptions as the Split-Trust configuration, with one exception, we assume that an encryption application may collude with the CSP client software to leak the user's data. The purpose of this configuration is to mitigate this threat by using multiple encryption modules, each containing a different encryption software application. We assume that though one of the encryption applications employed by the user may collude with the CSP client, not all of them do.

We have implemented this configuration, and provide a description of the implementation (section 4.3), as well as our security analysis of the implementation (section 4.4.2).

*4) Counter-CovertChannels:* The Counter-Collusion configuration protects the confidentiality of the user's data, when one of the encryption applications is colluding with the CSP client via overt channels, however, it doesn't guarantee the confidentiality of the data when they collude via covert channels. We thus designed, and implemented, the Counter-CovertChannels configuration to address this weakness.

The Counter-CovertChannels configuration, shown in Figure 4, also consists of multiple encryption modules, and a single CSP module, however the arrangement of the encryption modules, and consequently the encryption sequence, is different. It also differs from the two previous configurations in that we have two additional modules, an "Init" module used to coordinate data transfer to/from the encryption modules, as well as a "verification" module that's used to ensure that the ciphertexts produced by the two encryption modules are the same.

This configuration is designed with the same trust assumptions as the Counter-Collusion configuration. In addition to these assumptions, this configuration addresses the risk that a malicious encryption application may leverage covert channels to leak the user's data or cryptographic keys to a colluding CSP.

The main purpose of this configuration is to mitigate the threat of collusion by using multiple encryption modules, each using a different encryption application. We assume that though one of the encryption applications employed by the user may collude with the CSP client, directly or via covert channels, not all of them do.

We have implemented this configuration, and provide a

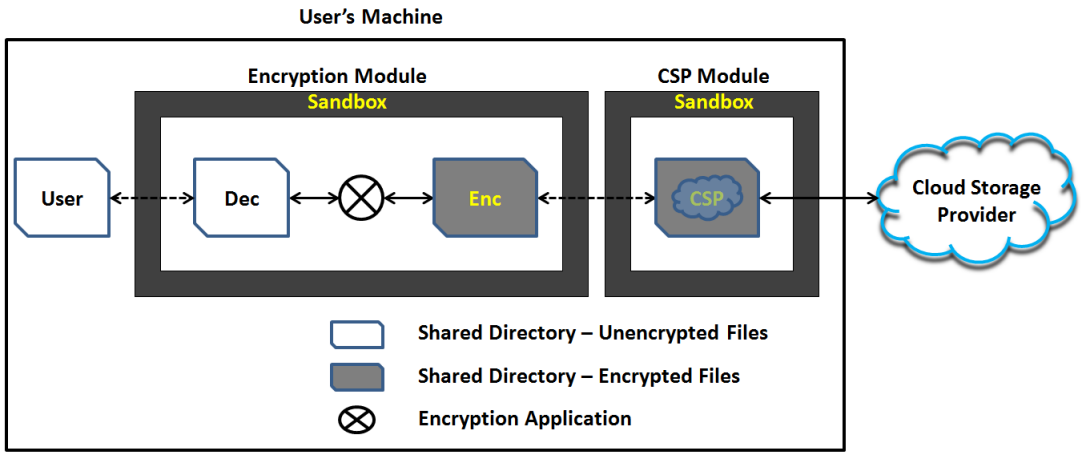


Figure 2. Split-Trust Configuration

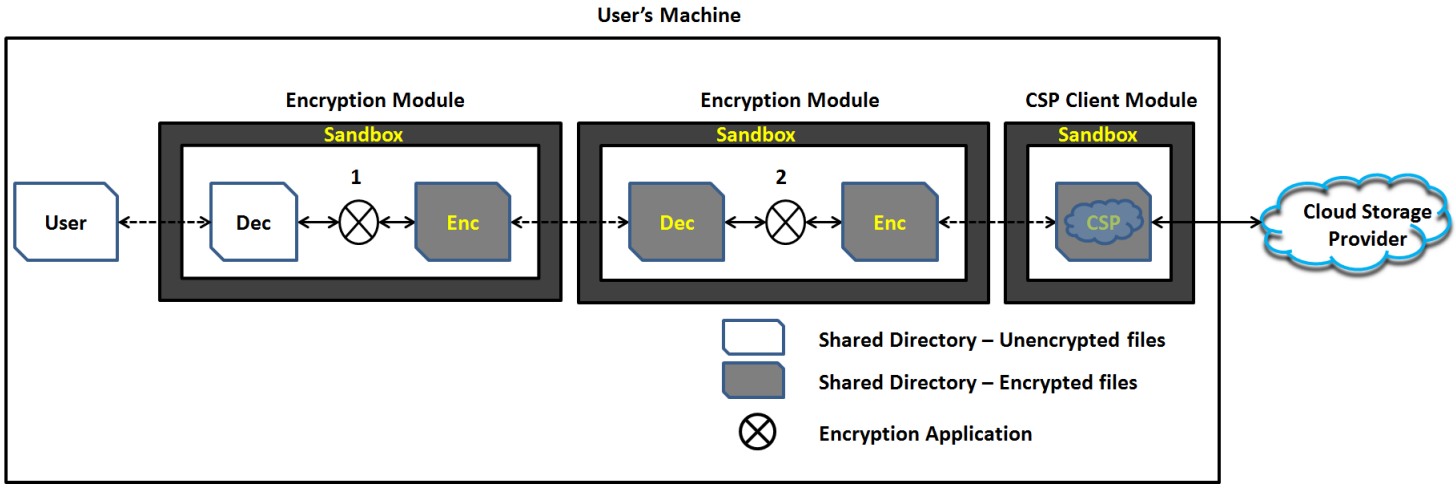


Figure 3. Counter-Collusion Configuration

description of the implementation (section 4.3), as well as our security analysis of the implementation (section 4.4.3).

### C. Implementation

We implemented the three configurations using the following technologies:

- Docker containers (https://docker.com)
- Incron (http://inotify.aiken.cz)

Docker containers provide the isolated environments in which the CSP client, and the encryption software, are installed and run. When creating a container, the docker client leverages the Linux kernel namespaces feature to create an isolated environment with its own process list, network device, user list, and filesystem. The docker containers are connected to each other by shared directories, mounted from the user's underlying filesystem.

The Incron system consists of a daemon and table manipulator, that allows a user to monitor a filesystem, and execute commands based on filesystem events. The user specifies the directory, and filesystem event to be monitored, as well as a command to execute when the event is detected. In our implementations, we install Incron in all the containers except the CSP client container, and use it to monitor the various directories on the path from the user's directory to the CSP client directory, triggering the appropriate sequence of actions when a change occurs.

In each configuration implementation, we also install the build-essential package (http://packages.ubuntu.com/trusty/build-essential) in each encryption container, which contains the development tools that enable us to build the third-party encryption application from source.

For the Split-Trust configuration (Figure 2), the process of syncing a file to the cloud proceeds as described below.

When the user creates, or updates, a file in the "User" directory, shared between the Encryption module and the user's system, Incron triggers the encryption process. The encryption software is invoked and the file is passed to it for encryption. The encrypted file is then sent to the CSP client module via the shared directory. The CSP client then begins the process of uploading it to the CSP's servers.

When a new encrypted file is received by the CSP client module from the CSP's server, the file is forwarded to the Encryption module via the shared interface. The decryption

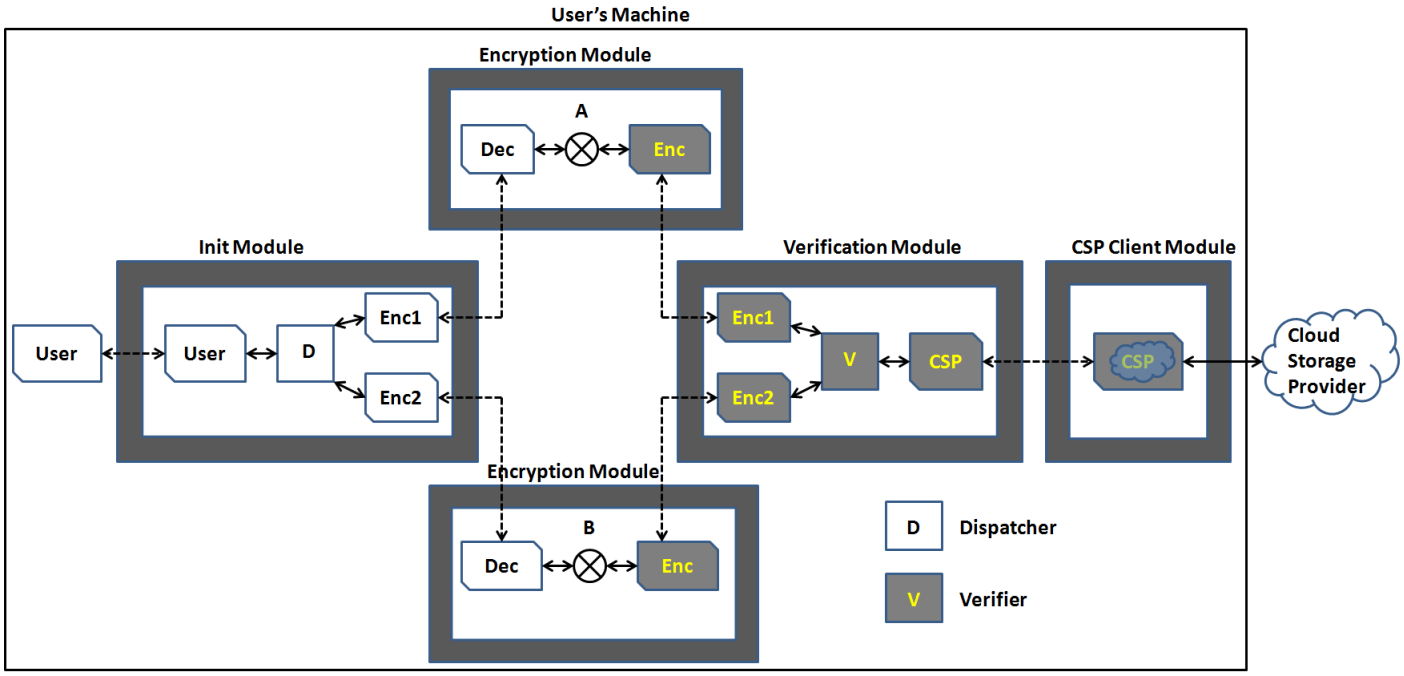


Figure 4. Counter-Covert Channels Configuration

process is triggered by Incron. The encryption application is invoked and the encrypted file passed to it. The encryption application then decrypts the file, and sends the decrypted file to the "User" directory.

In the implementation of the Counter-Collusion configuration (Figure 3), the process of syncing a file to the cloud proceeds in much the same way, as in the Split-Trust configuration. The difference is that you now have an additional encryption module through which the data is encrypted/decrypted before being sent to the cloud, or the user. Each encryption module uses a different encryption application, obtained from a different source, to encrypt/decrypt the data.

For the Counter-CovertChannels configuration (Figure 4), the process of syncing a file to the cloud proceeds as described below.

When the user creates or updates a file in the "User" directory, shared between the "Init" module and the user's system, a copy of the file is sent to the two directories that interface with the encryption modules. When the file is created in each of these directories, Incron triggers the encryption process in the respective encryption module. In each encryption module, the encryption software is invoked and the file is passed to it for encryption. The resulting encrypted file is then sent to the verification module via the shared directory between the encryption module and the verification module. The verification module, checks that the ciphertexts received from the two encryption modules are the same, since they both use the same encryption algorithm, key, and initialization vector to encrypt the plaintext file. If the contents of the two ciphertexts are the same, one copy of the encrypted file is then sent to the CSP client module via the directory shared between the verification module and the CSP client module. The CSP client then begins the process of uploading it to the CSP's servers.

When a new encrypted file is received by the CSP client module from the CSP's server, the encrypted file is sent to the verification module via the directory shared with the CSP client module. A copy of the file is then sent to the two directories that interface with the encryption modules. Each encryption module decrypts the file, and sends the plaintext copy to the "Init" module, via the directory shared between the encryption module and the "Init" module. The "Init" module then checks that both the plaintext files are the same, and forwards one copy to the "User" directory.

## V. Security Analysis

Table 1 highlights the trust assumptions for each configuration that we implemented.

### A. Split-Trust Configuration

The Split-Trust configuration is based on the assumption that the user's hardware and operating system, as well as the isolation mechanism are trusted, but software obtained from external parties, such as the CSP client software, and the encryption software are not. It also assumes that though each untrusted application may attempt to compromise the confidentiality of the user's data, and leak it to external parties, the untrusted applications do not collude.

As long as these assumptions hold, our design ensures that the confidentiality of the user's data is protected. It prevents the user's unencrypted data, and cryptographic keys from being leaked outside the user's device by the CSP client application, or the encryption software. This is acheived by isolating the CSP client from the rest of the user's system and the encryption environment, and allowing it to only access encrypted

| Configuration | Core Trusted Components | Trust Assumptions (Common) | Trust Assumptions (Config-Specific) |
|---|---|---|---|
| Conventional | Hardware and Operating System. | — | CSP client is trusted |
| Split-Trust | | Isolation Mechanism is trusted.<br>Encryption application correctly encrypts data.<br>Filesystem monitoring software doesn't collude with CSP client. | CSP client and Encryption application don't collude. |
| Counter-Collusion | | | CSP client and one Encryption application collude only via overt channels. |
| Counter-CovertChannel | | | CSP client and one Encryption application collude via overt and covert channels. |

Table I
CONFIGURATION TRUST ASSUMPTIONS

user data. The encryption application is also isolated, and its network access restricted. The Incron filesystem monitoring utility, as well as the development tools used to build the encryption application, are also isolated in the encryption container, with no network access.

For the convenience of the user, we provide scripts to setup the configuration on the user's device, and manage the synchronization of the encrypted/decrypted files between the user's directory and that of the CSP client software.

Current solutions require the user to either trust the CSP, or to trust both the CSP and a third party (in situations where the user employs a third-party software to encrypt the data before sending it to the cloud). In either case, the software obtained from the CSP is not restricted while running on the user's device. Our design, does not require the user to trust the CSP, since it restricts the CSP's client to accessing only encrypted data. However, it still requires the user to trust an external third-party, since the scripts implementing the configuration on the user's device, and managing the syncing process are obtained from an external party. Though the ideal situation would be one in which the user isn't required to trust any third-party software, we believe that our framework is still a viable solution, since the scripts we provide (Appendix A - C) are small, and could be easily audited by the user to confirm that it is implementing the configuration correctly and no malicious code is included. We therefore don't require the user to blindly trust our scripts. This contrasts with CSP client applications, which in most cases the user has no access to the source code. Even if they had access to the source code, it might be complex and large enough to make auditing it non-trivial. Another issue that is specific to our implementation, is that it requires the user to trust docker, which is provided by an external party, to not be malicious. Assuming a scenario where the providers of Docker are malicious, or it has some vulnerability, our current implementation would be vulnerable, since the isolation mechanism has access to the user's unencrypted data. However, since our design doesn't restrict us to using Docker to provide the implementation mechanism, we could provide a more secure implementation which does not rely on an externally obtained application to establish the isolated environments.

### *B. Counter-Collusion Configuration*

The Counter-Collusion configuration is based on the assumption that the user's hardware and operating system, as well as the isolation mechanism are trusted, but software obtained from external parties, such as the CSP client software, and the encryption software, are not. Also, unlike in the Split-Trust configuration where we assume non-collusion between the untrusted components, here we assume that some of these untrusted applications (specifically one of the encryption applications and the CSP client), may collude to compromise the confidentiality of the user's data. For this configuration we assume that the malicious encryption application and CSP client collude only via overt channels. The goal is to protect the confidentiality of the user's data.

In addition to addressing the same threats as the Split-Trust configuration, the Counter-Collusion configuration addresses the threat of a malicious encryption application leaking the user's data or cryptographic key to the CSP client, by embedding the information in the ciphertext, or weakly encrypting the data. As long as one of the encryption applications in this configuration, operates correctly, and uses a strong encryption algorithm, this configuration preserves the confidentiality of the user's data. However, this configuration does not guarantee confidentiality of the user's data in the scenario where the malicious encryption application uses covert channels to leak sensitive information to the CSP client.

This configuration also does not guarantee the confidentiality of the user 's data, in a scenario where either the filesystem monitoring tool (Incron in our implementation), or one of the development tools included in the build-essential package, is malicious and colludes with the CSP client. This is because these components are installed in all of the encryption containers, and may be able to access the user's unencrypted data, as well as the cryptographic keys. Therefore if either component is colluding with the CSP client, it would leak the user's information to the CSP client application.

As in the Split-Trust configuration, the scripts we provide (Appendix D), to implement this configuration on the user's device, and manage the syncing process, are small and could be easily audited by the user to confirm that they work correctly and no malicious code is included. We therefore don't require the user to blindly trust our scripts, and wrapper

code.

### C. Counter-Covert Channels Configuration

The Counter-CovertChannels configuration is based on the assumption that the user's hardware and operating system, as well as the isolation mechanism are trusted, but software obtained from external parties, such as the CSP client software, and the encryption software, are not. As in the Counter-Collusion configuration, we also assume that one of the encryption applications is malicious and colludes with the CSP client. However, for this configuration we assume that the malicious encryption application and CSP client may collude via both overt an covert channels.

For the convenience of the user, we provide scripts (Appendix E - O) to setup this configuration on the user's device, and manage the synchronization of the encrypted/decrypted files between the user's directory and that of the CSP client software. We also provide wrapper code (written in C) to facilitate the creation of a custom encryption application using a standard cryptographic library chosen by the user. This configuration requires the use of two encryption applications that produce the same cipher text, given the same input file and cryptographic key (and same initialization vector, if applicable). If the user can find two third-party encryption applications that achieve this, then there's no need for the user to create custom encryption applications using the wrapper code.

As in the previous configurations, the scripts we provide, to implement the Counter-Covert Channels configuration on the user's device, and manage the syncing process, as well as the encryption application wrapper code, are small and could be easily audited by the user to confirm that they work correctly and no malicious code is included. We therefore don't require the user to blindly trust our scripts, and wrapper code.

This configuration prevents the user's unencrypted data, and cryptographic key(s) from being leaked outside the user's device by the CSP client application, or the encryption software, if each is acting alone. This is acheived by isolating the CSP client from the rest of the user's system and the encryption environment, and allowing it to only access encrypted user data. The encryption application is also isolated, and its network access restricted. To mitigate the threat of collusion between the encryption application and the CSP client application, we employ multiple encryption modules, each running an encryption application composed from cryptographic libraries obtained from different parties. We then verify (in the verfication module) that the two encryption modules produce the same ciphertext. This protects against a malicious encryption application that adds extra data to the ciphertext before it's sent to the CSP client.

A malicious encryption application could also leak information about the user's data, or cryptographic key, to the CSP client via a timing covert channel. Shah et al. [17] show how data can be leaked covertly by modulating the timing of events that affect externally observable network traffic. It might not be practical for the malicious encryption application to leak the user's data this way, assuming the user is uploading a sizable amount of data. However, it could still leak the user's cryptographic key, or metadata associated with the user's files, which could provide useful information to a malicious CSP. Also if it's been designed to search for particular strings in the user's files, then leaking the data via a covert channel could be viable. The Counter-Covert Channels configuration mitigates the timing covert channel threat by verifying (in the verification module) that the same ciphertext file is received from each of the two encryption modules, and that they are both received within a small time window, before forwarding a copy to the CSP client application. While this prevents the malicious encryption application from arbitrarily choosing when to send files to the CSP client, it doesn't completely eliminate the timing channel, since the malicious encryption application could still leak some information by modulating the sending time of the encrypted file within the allowed window. We thus suggest that the user employ some logging mechanism to log the time taken by each encryption application to encrypt and send files, and to regularly audit the logs for any inconsistencies.

This configuration does not guarantee the confidentiality of the user's data, in a scenario where the filesystem monitoring tool (Incron in our implementation), is malicious and colludes with the CSP client. Though it provides the same guarantee of confidentiality for the scenario where one of the development tools in the build-essential package is colluding with the CSP client, as it does for the case of a malicious encryption application colluding with the CSP client.

## VI. Performance Evaluation

To understand the latency costs of our framework, we conducted experiments to measure the latency incurred when using the framework.

We performed the experiments using the Dropbox CSP client using the following setup:

- Two Desktop computers, M1 and M2, with the following specs
  - M1: OS- Ubuntu 14.04.1, Linux Kernel 3.13.0-35, CPU- Intel Core i7-2600 3.40GHz, 8GB RAM
  - M2: OS- CentOS 6.5, Linux Kernel 2.6.32-431, CPU- Intel Core i5-750 2.67GHz, 4GB RAM

  On each computer, we installed a Dropbox client application and linked it to a valid Dropbox account. We also setup the configuration to be evaluated, on each computer, using Docker v1.2.0, and Incron v0.5.10-1, and installed the Dropbox client in the CSP client container. The encryption applications we used are AESCRYPT v3.10 (for the Split-Trust and Counter-Collusion configurations), CCRYPT v1.10 (for the Counter-Collusion configuration), and two custom encryption applications based on the Cryptlib v3.43 and Libgcrypt v1.16.3 cryptographic libraries (for the Counter-CovertChannels configuration). The Wrapper code we wrote, to leverage these cryptographic libraries is included in Appendix (sections H - I)
- We used the testing application, made available by Drago et al. [18], to generate benchmarks, similar to those

| Configuration | Space Cost (no. of files) | Scripts Size (Lines of Code) |
|---|---|---|
| Conventional | 1x | N/A |
| Split-Trust | 2x | 136 |
| Counter-Collusion | 3x | 235 |
| Counter-CovertChannels | 6x | 633 |

Table V
CONFIGURATION COSTS

used in their work. The application also automates the experiment runs. However, in our evaluation, we record and analyse the end to end latency between two user devices that are linked to the same cloud storage account.

**Experimental Procedure**

For each configuration, we measured two aspects of the latency, the end to end delay, and the CSP delay.

- The end to end delay was measured from the point at which the file is created in the "User" directory on M1, to the point when it is received in the "User" directory on M2, both devices synced to the same CSP account.
- The CSP delay was measured from the point at which the file is created in the CSP client's directory on M1, to the point when it is received in the corresponding directory on M2.

During the experiment, the testing application generates the workloads, and sends them to the specified directory on M1, via FTP. For each scenario, the experiment was repeated 20 times, with 5 minute intervals between each experiment iteration.

**Observations**

It should be noted that when carrying out the measurements for each configuration (including the Conventional configuration), the two machines, M1 and M2, were on the same network, and Dropbox's LanSync feature was on. This means that for each configuration the time to retrieve the file, on the receiving end, was reduced, since the file was fetched directly from the sending host, rather than from the Dropbox servers.

From the results for each of the three configurations, indicated in tables II - IV, we observe that the CSP sync latency component comprises a large portion (>90%) of the end to end latency when using the framework.

Note that one of the benefits that we lose when using our framework, is the bandwidth (bytes transferred) and latency savings from deduplication. Since each file is encrypted before being sent to the Dropbox folder, when using our framework, the Dropbox client will transfer the whole file to the Dropbox servers. Whereas in the Conventional configuration, after dividing the file into chunks, the Dropbox client only transfers the chunks of a file that are unique (not located on Dropbox servers).

The storage cost, as well as the size of the scripts used, for each of the configurations, are indicated in Table V. We observe that the storage cost increases significantly from the Convential configuration to the Counter-CovertChannels configuration. This is the main cost of using our framework.

## VII. CONCLUSION

To enable users to leverage Cloud Storage Provider services, while protecting the confidentiality of their data, several solutions are currently available. These solutions allow the user to encrypt the data, and store all cryptographic keys locally, however they still require trusting the CSP to be benign.

This work presents a framework that facilitates the use of services offered by a Cloud Storage Provider without requiring the user to trust the CSP with access to its data. Our framework achieves this by isolating the CSP client application and encryption application from each other, and the rest of the system, and limiting their privileges. We have shown that our framework is versatile, and can be configured to satisfy different trust assumptions. We have also made available our scripts (https://github.com/uvasrg/NimboStratus) for the configurations we implemented, which are small in size and can be easily audited by the user. Our framework does not require the user to trust either the CSP or encryption application, nor the scripts that we provide. However it does require the user to trust the software implementing the isolation mechanism, which in this work we have assumed is benign.

In conclusion, our framework provides strong protection against the compromise of the data's confidentiality, assuming no collusion between the untrusted CSP and third-party applications. However, even in the presence of collusion (both overt and covert) between a malicious CSP client application and third-party application, our framework makes it harder, than current solutions, to compromise the confidentiality of the user's data.

## REFERENCES


[1] D. Wakabayashi, "Tim Cook says Apple to add security alerts for iCloud users," http://online.wsj.com/articles/tim-cook-says-apple-to-add-security-alerts-for-icloud-users-1409880977, September 2014.

[2] C. Brooks, "Cloud storage often results in data loss," http://www.businessnewsdaily.com/1543-cloud-data-storage-problems.html, October 2011.

[3] S. Byrne, "Microsoft OneDrive for business modifies files as it syncs," http://www.myce.com/news/microsoft-onedrive-for-business-modifies-files-as-it-syncs-71168/, April 2014.

[4] M. Borgmann and M. Waidner, *On the security of cloud storage services*. Fraunhofer-Verlag, 2012.

[5] R. Harvey, "Why client-side encryption is critical for cloud privacy," http://www.networkcomputing.com/why-client-side-encryption-is-critical-for-cloud-privacy/a/d-id/1234703, March 2014.

[6] A. Ferdowsi, "Yesterday's Authentication Bug," https://blog.dropbox.com/2011/06/yesterdays-authentication-bug/, June 2011.

[7] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Financial Cryptography and Data Security*, 2010.

[8] G. Danezis and B. Livshits, "Towards ensuring client-side computational integrity," in *ACM workshop on Cloud computing security*, 2011.

[9] S. Bugiel, S. Nürnberger, A.-R. Sadeghi, and T. Schneider, "Twin clouds: An architecture for secure cloud computing," in *Workshop on Cryptography and Security in Clouds*, 2011.

[10] A. Viswanathan and B. Neuman, "A survey of isolation techniques," in *Information Sciences Institute, University of Southern California*, 2009.

[11] N. Provos, "Improving host security with system call policies," in *USENIX Security*, vol. 3, 2003.

[12] T. Kim, N. Zeldovich, and R. Chandra, "Practical and effective sandboxing for non-root users," in *USENIX Annual Technical Conference*, 2013.

[13] H. Vijayakumar, J. Schiffman, and T. Jaeger, "Process firewalls: Protecting processes during resource access," in *ACM European Conference on Computer Systems*, 2013.

| Files | File Size | End to End Latency (s) | CSP Sync latency (s) | CSP Sync latency component (%) |
|---|---|---|---|---|
| 1 | 100KB | 5.822 | 5.732 | 98.45 |
| 1 | 1MB | 6.686 | 6.511 | 97.38 |
| 10 | 100KB | 15.981 | 15.844 | 99.14 |
| 100 | 10KB | 81.553 | 81.348 | 99.75 |

Table II
SPLIT-TRUST CONFIGURATION - END TO END VS. CSP SYNC LATENCY

| Files | File Size | End to End Latency (s) | CSP Sync latency (s) | CSP Sync latency component (%) |
|---|---|---|---|---|
| 1 | 100KB | 5.920 | 5.789 | 97.78 |
| 1 | 1MB | 6.259 | 5.997 | 95.79 |
| 10 | 100KB | 16.831 | 16.610 | 98.69 |
| 100 | 10KB | 78.686 | 78.005 | 99.13 |

Table III
COUNTER-COLLUSION CONFIGURATION - END TO END VS. CSP SYNC LATENCY

| Files | File Size | End to End Latency (s) | CSP Sync latency (s) | CSP Sync latency component (%) |
|---|---|---|---|---|
| 1 | 100KB | 5.856 | 5.712 | 97.54 |
| 1 | 1MB | 6.452 | 6.194 | 96.00 |
| 10 | 100KB | 16.638 | 16.231 | 97.55 |
| 100 | 10KB | 85.211 | 83.983 | 98.56 |

Table IV
COUNTER-COVERTCHANNELS CONFIGURATION - END TO END VS. CSP SYNC LATENCY


[14] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, 1975.

[15] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments." *Usenix Symposium on Network Systems Design and Implementation*, vol. 8, 2008.

[16] A. Blankstein and M. J. Freedman, "Automating isolation and least privilege in web services," in *IEEE Symposium on Security and Privacy (SP)*, 2014.

[17] G. Shah, A. Molina, M. Blaze *et al.*, "Keyboards and Covert channels." in *USENIX Security*, 2006.

[18] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking Personal Cloud Storage," in *ACM Conference on Internet Measurement*, 2013.

## APPENDIX

### A. Split-Trust Configuration Setup script (split-trust_config.sh)

```
#! /bin/bash

# The USER_DIR directory is created on the user's filesystem, and mounted onto the
# ENC_MODULE docker container (with read and write permissions). This is the
# directory through which the user updates (plaintext) files to be synced to the cloud,
# and receives updated (plaintext) files from the cloud.
# The ENC_CSP_INTERFACE directory is also created on the user's filesystem, and
# is mounted onto the ENC_MODULE and CSP_MODULE docker containers
# (with read and write permissions). This is the directory through which the two docker
# containers exchange encrypted files.
# These (USER_DIR and ENC_CSP_INTERFACE) are the only two directories, on
# the user's filesystem, that applications running within the docker containers are allowed to
# access.

    BASE_DIR=~/framework_space/
    USER_DIR=${BASE_DIR}/plain_dir/
    ENC_CSP_INTERFACE=${BASE_DIR}/enc-csp_dir/

# DECRYPTED_DIR is the name given to the USER_DIR directory, within the
# ENC_MODULE container. It is where the user's plaintext files are placed, before being
# encrypted and sent to the CSP, or after being received from the CSP, and decrypted.
# ENCRYPTED_DIR is the name given to the ENC_CSP_INTERFACE directory, within
# the ENC_MODULE container. It is where the user's ciphertext files are placed, after being
# encrypted.
# The TMP_ENC_DIR and TMP_DEC_DIR directories, are used to temporarily hold the files
# that are being encrypted/decrypted.
# The SCRIPTS_DIR directory is where the scripts that coordinate the encryption/decryption
# process are located.
# All of these directories are created in the ENC_MODULE container, and,
# (apart from the DECRYPTED_DIR and ENCRYPTED_DIR directories, which are shared)
# can only be accessed by applications running in this container.

    DECRYPTED_DIR=/decrypted/
    ENCRYPTED_DIR=/encrypted/
    TMP_ENC_DIR=/enc_int/
    TMP_DEC_DIR=/dec_int/
    SCRIPTS_DIR=/scripts/
    ENC_MODULE="automated-enc_module"
    IMAGE="framework:enc_module"

# CSP_DIR is the name given to the ENC_CSP_INTERFACE directory, within the
# CSP_MODULE container. This is the directory (shared with the ENC_MODULE container)
# which the CSP client monitors, and syncs with the CSP infrastructure.
# Note that the CSP client application running in the CSP_MODULE container has access only to
# the ENC_CSP_INTERFACE directory on the user's file system.

    CSP_DIR=/csp/
    CSP_MODULE="automated-csp_module"

    Enc_Image_Setup () {
# This line creates a docker container (with ubuntu as the operating system), running in the background (-d),
# with a pseudo terminal attached (-t), and the capability to receive input from the keyboard (-i)
    docker run -d -i -t --name="ENC_IMAGE" ubuntu

# The following line runs a shell in the created docker container, and executes commands to:
```

```
# - create the temp directories used during the encryption/decryption process, as well as the directory for holding
# the scripts
# - update the list of Ubuntu repository packages ("apt-get update"), which is used when installing packages
# from the Ubuntu repositories via the apt-get install <package> command. The command "apt-get update"
# doesn't install anything in the container.
# Here, we are trusting that when this command is executed, a connection is established to the Ubuntu repositories,
# and the list of packages, with the correct signature for each package, is retrieved from the repository.
# - install the build-essential package from Ubuntu's repository ("apt-get -y install build-essential"), that is needed
# for building applications from source. We need this package to build the encryption application. This package
# consists of references to packages needed for building software on Ubuntu, including compilers (gcc and g++),
# GNU C library, and the MAKE utility.
# NOTE: If any of these packages is malicious, then it has access to the user's unencrypted files, and may be
# able to access the user's crypto key. Since we are assuming non-collusion for this configuration, this isn't an issue.
# The encryption container has no network access, and these packages are not installed in the CSP client container.
# Thus there's no way to leak the user's data/crypto key.
# - install the incron filesystem monitoring utility ("apt-get install incron")
# NOTE: This is a third party application, that has access to the user's unencrypted files, and also may be able to access
# the user's crypto key. However, since the encryption container has no network access, and this application isn't
# installed in the CSP client container, even if this application is malicious, it has no way to leak the user's data under
# this configuration.
# - enable the "root" user to use incron within the container ("echo 'root' > /etc/incron.allow")
# - create the filesystem monitoring rules for incron, and save them (in /var/spool/incron/root).
# The rules specify which directory to monitor (e.g. $DECRYPTED_DIR), which event to monitor for
# (e.g. IN_CLOSE_WRITE), and what action to take
# (e.g. $SCRIPTS_DIR/encrypt_script.pl $% $'@'$'#' \ $ENCRYPTED_DIR $TMP_ENC_DIR
# $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' – this calls the "encrypt_script" (which we provide, and the user can audit),
# and passes it the name of the event {$%} that triggered incron, the url {$@$#} of the file that triggered
# incron, the names of the directories used during the encryption/decryption process, as well as the url of the keyfile
# containing the encryption key/password).

    docker exec ENC_IMAGE sh -c "mkdir $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR; \
    apt-get update; apt-get -y install build-essential; apt-get install incron; \
    echo 'root' > /etc/incron.allow; \

    echo '$DECRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/encrypt_script.pl $% $'@'$'#' \
    $ENCRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root; \

    echo '$ENCRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/decrypt_script.pl $% $'@'$'#' \
    $DECRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root"

# The following three lines, stop the ENC_IMAGE container, save its image, and then delete the container.
# NOTE: We required network access to download and install the components to build the encryption application,
# however our security requirements are that the encryption container should have no network access. We therefore
# delete this container which was built with network access, and then later (in the Enc_Mod_Setup function) use the
# saved image (which has the libraries/tools require to build the encryption application) to create an encryption container
# that has no network access.

    docker stop ENC_IMAGE
    docker commit ENC_IMAGE $IMAGE
    docker rm ENC_IMAGE
    }

    Enc_Mod_Setup () {
# This line creates the encryption container, from the previously saved image, with the two interface directories,
# USER_DIR and ENC-CSP_INTERFACE mounted in the container (-v), and with no network access (–net=none).

    docker run -d -i -t --net=none -v $1:$DECRYPTED_DIR -v $2:$ENCRYPTED_DIR --name=$3 $IMAGE

```

```
114) # The following line runs a shell in the created encryption container, and executes commands to:
115) # - generate a file containing random data ("dd if=/dev/urandom of=$SCRIPTS_DIR/keyfile.data bs=1024 count=100")
116) # - generate a hash of the "random data" file, and store the hash value in a separate file
117) # ("md5sum $SCRIPTS_DIR/keyfile.data | cut -d ' ' -f1 > $SCRIPTS_DIR/keyfile"). We use this hash value as the
118) # password/key for encryption/decryption, depending on the requirements of the encryption application selected by the
     user.
119) # NOTE: This 32 byte password/key, is generated randomly each time this configuration setup script is run
120) # (making it very hard to guess), and is used by default for encryption/decryption. The user can modify this value, at
     any
121) # time, by editing the "keyfile" file, located in the scripts directory. In order to share files with multiple devices, each
     running
122) # this configuration, the user will need to # ensure that each encryption container has the same copy of the "keyfile".
     Thus
123) # the user needs to find a way to securely transfer one of the "keyfile" copies to the encryption containers on the other
     devices.
124)
125)     docker exec $3 sh -c "dd if=/dev/urandom of=$SCRIPTS_DIR/keyfile.data bs=1024 count=100; \
126)     md5sum $SCRIPTS_DIR/keyfile.data | cut -d ' ' -f1 > $SCRIPTS_DIR/keyfile"
127)     }
128)
129)     CSP_Mod_Setup () {
130) # This line creates the CSP client container, with the "ENC-CSP_INTERFACE" directory mounted in the container.
131) # This container has network access enabled, so that the CSP client can communicate with the CSP servers.
132)     docker run -d -i -t -v $1:$CSP_DIR --name=$2 ubuntu
133)     docker exec $2 apt-get update
134)     }
135)
136) # We start by creating the "Base" directory, and the two shared directories.
137)     mkdir ${BASE_DIR} ${USER_DIR} ${ENC_CSP_INTERFACE}
138)
139) # We then create, and save, a template for the Encryption container to be built.
140)     Enc_Image_Setup
141)
142) # We then build the Encryption container, with no network access. The Enc_Mod_Setup function takes in 3 parameters:
143) # 2 names of the interface directories through which the encryption container receives data to be encrypted, and
144) # delivers encrypted data. 1 name of the encryption container
145)     Enc_Mod_Setup ${USER_DIR} ${ENC_CSP_INTERFACE} ${ENC_MODULE}
146)
147) # Finally, we build the CSP client container. The CSP_Mod_Setup function takes in 2 parameters:
148) # 1 name of the interface directory through which the CSP module receives data from, or to be synced to, the CSP,
149) # and 1 name of the CSP module container.
150)     CSP_Mod_Setup ${ENC_CSP_INTERFACE} ${CSP_MODULE}
```

*B. Split-Trust Configuration Encryption Script (encrypt_script.pl)*

```
1
2  #! /usr/bin/perl
3
4  use warnings;
5  use strict;
6
7  use File::Compare;
8  use File::Basename;
9  use File::Copy;
10
11 # The following variables are initialised using values passed from
12 # the configuration setup script (split-trust_config.sh)
13 # - The variable "url" is assigned the full path of the file
14 # that triggered this script.
15 # - The variable "encrypted" is assigned the full path of the directory
16 # where encrypted files are to be placed.
17 # - The "tmpEnc" and "tmpDec" directories are assigned the full paths
18 # of the the directories used to temporarily hold files during the
19 # encryption/decryption process.
20 # - The "passFile" variable is assigned the full path of the file
21 # containing the password/key used for encryption
22
23 my $url = $ARGV[1];
24 my $encrypted = $ARGV[2];
25 my $tmpEnc = $ARGV[3];
26 my $tmpDec = $ARGV[4];
27 my $passFile = $ARGV[5];
28
29 sub Encrypt{
30
31 # This function should be modified, to suit the syntax
32 # of the encryption application being used.
33 # It receives the url of the file to be encrypted, its name,
34 # and the encryption password/key.
35 # We then specify the name of the output file to be created
36 # after encryption(path + extension), call the encryption
37 # application, and return the full path to the encrypted file.
38 # Note that the encrypted file is placed in the tmpEnc
39 # directory.
40 # This helps the "decrypt_script" distinguish between the case
41 # where a file entering the "Enc" directory, is from the CSP and
42 # needs to be decrypted, or is from the user and has just been
43 # encrypted. We need to distinguish these two cases in order to
44 # avoid getting into an infinite loop of encryption/decryption over
45 # the same file. After this function returns, the encrypted file is
46 # copied to the encrypted directory.
47
48         my $fileURL = $_[0];
49         my $fileName = $_[1];
50         my $pWord = $_[2];
51         my $ext = '.aes';
52         my $output = "$tmpEnc"."$fileName"."$ext";
53         `aescrypt -e -p $pWord -o $output $fileURL`;
54         return $output;
55 }
56
```

```
57 # This script monitors the "Dec" directory which holds the user's
58 # plaintext files. When the user uploads a file to "USER_DIR"
59 # which interfaces with the encryption container, this script is
60 # triggered, by the matching rule specified in the incrontab file
61 # (/var/spool/incron/root), and this matching rule passes the url
62 # of the file to this script.
63 # Note that the rules, specifiying the directories to be monitored
64 # by incron, the events to monitor, and the actions to take when
65 # the events occur, are created during the configuration setup
66 # phase (split-trust_config.sh -- specifically in the Enc_Image_Setup()
67 # function)
68
69 chomp($url);
70
71 # we extract the file name from the url
72 my $fname = fileparse($url);
73 my $compareFile = $tmpDec . $fname;
74
75 # We check if the file that triggered the script has a similar copy
76 # in the tmpDec directory. If a similar copy exists in the tmpDec directory,
77 # it means that the file should not be encrypted, since it has just
78 # been decrypted. When the "decrypt_script", in charge of decryption, decrypts
79 # a file, it places a copy of the decrypted plaintext file in the tmpDec
80 # directory, before copying the file to the user's directory. This helps
81 # distinguish between the case where a file entering the "Dec" directory
82 # (which triggers this script), is from the CSP and has just been decrypted, or
83 # is from the user and needs to be encrypted before being sent to the CSP.
84 # We need to distinguish these two cases in order to avoid getting
85 # into an infinite loop of encryption/decryption over the same file.
86
87 my $fileCheck = compare($url, $compareFile);
88 if ($fileCheck == 0){
89
90 # if a similar copy of the file exists in the tmpDec directory, it means the
91 # received file has just been decrypted, and shouldn't be encrypted again.
92 # we delete the copy of the file in the tmpDec directory, and exit. There's
93 # nothing else to do.
94
95         unlink $compareFile;
96 } else{
97
98 # otherwise, the file needs to be encrypted, and we proceed as follows:
99 # we open the file containing the encryption password/key, extract the password/key,
100 # and close the file.
101
102         open(PASSFILE, "<$passFile") || die ("failed to open $passFile\n");
103         my $passWord = <PASSFILE>;
104         chomp($passWord);
105         close(PASSFILE);
106
107 # we then pass the url of the file to be encrypted, the filename, and the encryption
108 # password/key to the encryption function. The encryption function encrypts the
109 # file, then returns the full path to the encrypted file.
110 # The encrypted file is then copied to the encrypted directory, and this script's
111 # work is done.
112
113         my $outFile = Encrypt($url, $fname, $passWord);
114         copy($outFile, $encrypted);
```

```
115 }
```

*C. Split-Trust Configuration Decryption Script (decrypt_script.pl)*

```
#! /usr/bin/perl

use warnings;
use strict;
use File::Compare;
use File::Basename;
use File::Copy;

# The following variables are initialised using values passed from
# the configuration setup script (split-trust_config.sh)
# - The variable "url" is assigned the full path of the file that
# triggered this script.
# - The variable "decrypted" is assigned the full path of the directory
# where encrypted files are placed
# - The "tmpEnc" and "tmpDec" directories are assigned the full paths
# of the the directories used to temporarily hold files during the
# encryption/decryption process.
# - The "passFile" variable is assigned the full path of the file
# containing the password/key used for decryption

my $url = $ARGV[1];
my $decrypted = $ARGV[2];
my $tmpEnc = $ARGV[3];
my $tmpDec = $ARGV[4];
my $passFile = $ARGV[5];

sub Decrypt{

# This function should be modified, to suit the syntax of the
# encryption application being used. It receives the url of
# the file to be decrypted, its name, and the decryption password/key.
# We then specify the name of the output file to be created after
# decryption(path), call the encryption application, and return the
# full path to the decrypted file.
# Note that the decrypted file is placed in the tmpDec directory.
# This helps the "encrypt_script" distinguish between the case where a
# file entering the "Dec" directory, is from the user and needs to be
# encrypted, or is from the CSP and has just been decrypted. We need
# to distinguish these two cases in order to avoid getting into an
# infinite loop of encryption/decryption over the same file. After this
# function returns, the decrypted file is copied to the "Dec" directory.

        my $fileURL = $_[0];
        my $fileName = $_[1];
        my $pWord = $_[2];
        my $output = "$tmpDec"."$fileName";
        $output =~ s/.aes//;
        `aescrypt -d -p $pWord -o $output $fileURL`;
        return $output;
}

# This script monitors the "Enc" directory which holds the user's
# ciphertext files. When a file uploaded by the user is encrypted,
# and placed in this directory, or an encrypted file is received
# from the CSP, this script is triggered, by the matching rule
# specified in the incrontab file (/var/spool/incron/root), and
```

```
57 # this matching rule passes the url of the file to this script.
58 # Note that the rules, specifiying the directories to be monitored by
59 # incron, the events to monitor, and the actions to take when the
60 # events occur, are created during the configuration setup phase
61 # (split-trust_config.sh -- specifically in the Enc_Image_Setup() function)
62
63 chomp($url);
64
65 # we extract the file name from the url
66 my $fname = fileparse($url);
67 my $compareFile = $tmpEnc . $fname;
68
69 # We check if the file that triggered the script has a similar copy
70 # in the "tmpEnc" directory. If a similar copy exists in the tmpEnc
71 # directory, it means that the file should not be decrypted, since
72 # it has just been encrypted. When the "encrypt_script", in charge
73 # of encryption, encrypts a file, it places a copy of the encrypted
74 # file in the tmpEnc directory, before copying the file to the "Enc"
75 # directory. This helps distinguish between the case where a file
76 # entering the "Enc" directory(which triggers this script), is from
77 # the CSP and needs to be decrypted, or is from the user and has
78 # just been encrypted. We need to distinguish these two cases in order
79 # to avoid getting into an infinite loop of encryption/decryption over
80 # the same file.
81
82 my $fileCheck = compare($url, $compareFile);
83
84 if ($fileCheck == 0){
85
86 # if a similar copy of the file exists in the tmpEnc directory, it means
87 # the received file has just been encrypted, and shouldn't be decrypted.
88 # we delete the copy of the file in the tmpEnc directory, and exit. There's
89 # nothing else to do.
90
91         unlink $compareFile;
92 } else{
93
94 # otherwise, the file needs to be decrypted, and we proceed as follows:
95 # we open the file containing the decryption password/key, extract the
96 # password/key, and close the file.
97
98         open(PASSFILE, "<$passFile") || die ("failed to open $passFile\n");
99         my $passWord = <PASSFILE>;
100        chomp($passWord);
101        close(PASSFILE);
102
103 # we then pass the url of the file to be decrypted, the filename, and
104 # the decryption password/key to the decryption function. The decryption
105 # function decrypts the file, then returns the full path to the decrypted
106 # file. The decrypted file is then copied to the decrypted directory,
107 # and this script's work is done.
108
109        my $outFile = Decrypt($url, $fname, $passWord);
110        copy($outFile, $decrypted);
111 }
```

*D. Counter-Collusion Configuration Setup script (counter-collusion_config.sh)*

```
#! /bin/bash

# The USER_DIR directory is created on the user's filesystem, and mounted onto the ENC_MODULE docker
# container (with read and write permissions). This is the directory through which the user updates (plaintext) files
# to be synced to the cloud, and receives updated (plaintext) files from the cloud.
# The ENC1_ENC2_INTERFACE directory is also created on the user's filesystem, and is mounted onto the
# ENC_MODULE and ENC_MODULE2 docker containers (with read and write permissions). This is the
# directory through which these two docker containers exchange encrypted files.
# The ENC2_CSP_INTERFACE directory is also created on the user's filesystem, and is mounted onto the
# ENC_MODULE and CSP_MODULE docker containers (with read and write permissions). This is the
# directory through which these two docker containers # exchange encrypted files. The USER_DIR,
# ENC1_ENC2_INTERFACE, and ENC2_CSP_INTERFACE directories, are the only directories, on
# the user's filesystem, that applications running within the docker containers are allowed to access.

    BASE_DIR=~/framework_space/
    USER_DIR=${BASE_DIR}/plain_dir/
    ENC1_ENC2_INTERFACE=${BASE_DIR}/enc1-enc2_dir/
    ENC2_CSP_INTERFACE=${BASE_DIR}/enc2-csp_dir/

# DECRYPTED_DIR is the name given to the USER_DIR directory, within the ENC_MODULE container.
# It is where the user's plaintext files are placed, before being encrypted and sent to ENC_MODULE2,
# or after being received from ENC_MODULE2, and decrypted.
# ENCRYPTED_DIR is the name given to the ENC1_ENC2_INTERFACE directory, within the
# ENC_MODULE container. It is where the user's ciphertext files are placed, after being encrypted
# by the encryption application in ENC_MODULE. In the ENC_MODULE2 container, the
# ENC1_ENC2_INTERFACE directory is mounted as DECRYPTED_DIR, and the
# ENC2_CSP_INTERFACE directory is mounted as ENCRYPTED_DIR. In both containers,
# ENC_MODULE and ENC_MODULE2, the TMP_ENC_DIR and TMP_DEC_DIR directories, are used to
# temporarily hold the files that are being encrypted/decrypted. The SCRIPTS_DIR directory is where the scripts
# that coordinate the encryption/decryption process are located.
# NOTE: The applications running in each of these containers can only access the directories(and files)in their
# respective containers. Apart from being able to access (read and write) the three directories, USER_DIR,
# ENC1_ENC2_INTERFACE, and ENC2_CSP_INTERFACE, that are mounted from the user's filesystem,
# the applications running in each of these containers have no access to any other part of the user's filesystem.

    DECRYPTED_DIR=/decrypted/
    ENCRYPTED_DIR=/encrypted/
    TMP_ENC_DIR=/enc_int/
    TMP_DEC_DIR=/dec_int/
    SCRIPTS_DIR=/scripts/
     ENC_MODULE="automated-enc_module"
    ENC_MODULE2="automated-enc_module2"
    IMAGE="framework:enc_module"

# CSP_DIR is the name given to the ENC2_CSP_INTERFACE directory, within the CSP_MODULE container.
# This is the directory (shared with the ENC_MODULE2 container) which the CSP client monitors, and syncs with the
# CSP infrastructure.
# Note that the CSP client application running in the CSP_MODULE container has access only to the
# ENC2_CSP_INTERFACE directory on the user's file system.

    CSP_DIR=/csp/
    CSP_MODULE="automated-csp_module"

    Enc_Image_Setup () {

# This line creates a docker container (with ubuntu as the operating system), running in the background (-d),
```

```
# with a pseudo terminal attached (-t), and the capability to receive input from the keyboard (-i)

    docker run -d -i -t --name="ENC_IMAGE" ubuntu

# The following line runs a shell in the created docker container, and executes commands to:
# - create the temp directories used during the encryption/decryption process, as well as the directory for holding
# the scripts
# - update the list of Ubuntu repository packages ("apt-get update"), which is used when installing packages from
# the Ubuntu repositories via the apt-get install <package> command. The command "apt-get update" doesn't install
# anything in the container. Here, we are trusting that when this command is executed, a connection is established to
# the Ubuntu repositories, and the list of packages, with the correct signature for each package, is retrieved from
# the repository
# - install the build-essential package from Ubuntu's repository ("apt-get -y install build-essential"), that is needed for
# building applications from source. We need this package to build the encryption application. This package consists of
# references to packages needed for building software on Ubuntu, including compilers (gcc and g++), GNU C library,
# and the MAKE utility
# NOTE: We assume that these packages are not malicious, since they are obtained from the Ubuntu repository, and we
# are trusting our operating system, and by extension the OS provider. However, if any of these packages is malicious,
# and it colludes with the CSP client (under this configuration, we assume some malicious components may collude),
then
# this would represent a serious vulnerability. Since, these packages are installed in each of the encryption containers,
# and thus would be able to access the user's unencrypted data, as well as cryptographic keys.

# - install the incron filesystem monitoring utility ("apt-get install incron")
# NOTE: This is a third party application, that has access to the user's unencrypted files, and also may be able to access
# the user's crypto keys. Again, under this configuration, we assume some malicious components may collude, therefore
# if this filesystem monitoring tool is malicious, and colludes with the CSP client application, it would be able to leak
the
# the user's data, or cryptographic keys. We assume that this application is trusted not to collude with the CSP client
# application.
# - enable the "root" user to use incron within the container ("echo 'root' > /etc/incron.allow")
# - create the filesystem monitoring rules for incron, and save them (in /var/spool/incron/root).
# The rules specify which directory to monitor (e.g. $DECRYPTED_DIR), which event to monitor for
# (e.g. IN_CLOSE_WRITE), and what action to take
# (e.g. $SCRIPTS_DIR/encrypt_script.pl $% $'@'$'#' $ENCRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR
# $SCRIPTS_DIR/keyfile' – this calls the "encrypt_script" (which we provide, and the user can audit), and passes it
# the name of the event {$%} that triggered incron, the url {$@$#} of the file that triggered incron, the names of the
# directories used during the encryption/decryption process, as well as the url of the keyfile containing the encryption
# key/password).

    docker exec ENC_IMAGE sh -c "mkdir $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR; \
    apt-get update; apt-get -y install build-essential; apt-get install incron; \
    echo 'root' > /etc/incron.allow; \

    echo '$DECRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/encrypt_script.pl $% $'@'$'#' \
    $ENCRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root; \

    echo '$ENCRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/decrypt_script.pl $% $'@'$'#' \
$DECRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root"

# The following three lines, stop the ENC_IMAGE container, save its image, and then delete the container.
# NOTE: We required network access to download and install the components to build the encryption application,
# however our security requirements are that the encryption container should have no network access. We therefore
# delete this container which was built with network access, and then later (in the Enc_Mod_Setup function) use the
# saved image (which has the libraries/tools require to build the encryption application) to create an encryption container
# that has no network access.

    docker stop ENC_IMAGE
```

```
113)     docker commit ENC_IMAGE $IMAGE
114)     docker rm ENC_IMAGE
115)     }
116)
117)     Enc_Mod_Setup () {
118)
119) # This line creates the encryption container, from the previously saved image, with the two interface directories, specified
120) # by the user, mounted in the container (-v), and with no network access (–net=none).
121) # The following line runs a shell in the created encryption container, and executes commands to:
122) # - generate a file containing random data ("dd if=/dev/urandom of=$SCRIPTS_DIR/keyfile.data bs=1024 count=100")
123) # - generate a hash of the "random data" file, and store the hash value in a separate file
124) # ("md5sum $SCRIPTS_DIR/keyfile.data | cut -d ' ' -f1 > $SCRIPTS_DIR/keyfile"). We use this hash value as the
125) # password/key for encryption/decryption, depending on the requirements of the encryption application selected by the
     user.
126) # NOTE: This 32 byte password/key, is generated randomly each time this configuration setup script (and this function)
     is
127) # run, and is used by default for encryption/decryption. The goal is to generate a default password/key for the user that
     is
128) # hard to guess, unique for each encryption container, and can be easily customized by the user.
129) # The user can modify this value, at any time, by editing the "keyfile" file, located in the scripts directory. Since there
     are two
130) # encryption containers in this configuration, each has a different keyfile.data generated, and thus a different keyfile
     value.
131) # In order to share files with multiple devices, each running this configuration, the user will need to ensure that the same
     copy
132) # of the two keyfiles (ENC_MODULE and ENC_MODULE2) generated on one device, are used in the corresponding
133) # encryption containers, in each of the other devices. Thus the user needs to find a way to securely transfer copies of
     the two
134) # keyfiles to the corresponding encryption containers on the other devices.
135)
136)     docker run -d -i -t --net=none -v $1:$DECRYPTED_DIR -v $2:$ENCRYPTED_DIR --name=$3 $IMAGE
137)     docker exec $3 sh -c "dd if=/dev/urandom of=$SCRIPTS_DIR/keyfile.data bs=1024 count=100; \
138)     md5sum $SCRIPTS_DIR/keyfile.data | cut -d ' ' -f1 > $SCRIPTS_DIR/keyfile"
139)     }
140)
141)     CSP_Mod_Setup () {
142)
143) # This line creates the CSP client container, with the "ENC2-CSP_INTERFACE" directory mounted in the container.
144) # This container has network access enabled, so that the CSP client can communicate with the CSP servers.
145)
146)     docker run -d -i -t -v $1:$CSP_DIR --name=$2 ubuntu
147)     docker exec $2 apt-get update
148)     }
149)
150) # We start by creating the "Base" directory, and the three shared directories.
151)     mkdir ${BASE_DIR} ${USER_DIR} ${ENC1_ENC2_INTERFACE} ${ENC2_CSP_INTERFACE}
152)
153) # We then create, and save, a template for the Encryption containers to be built.
154)     Enc_Image_Setup
155)
156) # We build the two Encryption containers, with no network access. The Enc_Mod_Setup function takes in 3 parameters:
157) # - the names of the 2 interface directories through which the encryption container receives data to be encrypted, and
158) # delivers encrypted data.
159) # - the name of the encryption container
160)     Enc_Mod_Setup ${USER_DIR} ${ENC1_ENC2_INTERFACE} ${ENC_MODULE}
161)     Enc_Mod_Setup ${ENC1_ENC2_INTERFACE} ${ENC2_CSP_INTERFACE} ${ENC_MODULE2}
162)
163) # Finally, we build the CSP client container. The CSP_Mod_Setup function takes in 2 parameters:
```

```
164) # - the name of the interface directory through which the CSP client container receives data from, or to be synced to,
     the CSP
165) # - the name of the CSP client container
166)       CSP_Mod_Setup ${ENC2_CSP_INTERFACE} ${CSP_MODULE}
```

*E. Counter-CovertChannels Configuration Setup script (counter-covertchannels_config.sh)*

```
#! /bin/bash

# NOTE: The applications running in each of the five containers, INIT_MODULE, ENC_MODULE,
# ENC_MODULE2, VERIFICATION_MODULE, and the CSP_MODULE, can only access the
# directories(and files)in their respective containers. Apart from being able to access
# (read and write) the directories, USER_DIR, INIT_ENC1_INTERFACE,
# INIT_ENC2_INTERFACE, ENC1_VER_INTERFACE, ENC2_VER_INTERFACE, and
# VER_CSP_INTERFACE, that are mounted from the user's filesystem, the applications running in each
# of these containers have no access to any other part of the user's filesystem.

    BASE_DIR=~/framework_space/
    USER_DIR=${BASE_DIR}/plain_dir/
    INIT_ENC1_INTERFACE=${BASE_DIR}/init-enc1_dir/
    INIT_ENC2_INTERFACE=${BASE_DIR}/init-enc2_dir/
    VER_CSP_INTERFACE=${BASE_DIR}/ver-csp_dir/
    ENC_MODULE="automated-enc_module"
    ENC_MODULE2="automated-enc_module2"
    CSP_MODULE="automated-csp_module"
    VERIFICATION_MODULE="automated-ver_module"
    INIT_MODULE="automated-init_module"
    ENC_IMAGE="framework:enc_module"
    AUX_IMAGE="framework:aux_module"
    DECRYPTED_DIR=/decrypted/
    ENCRYPTED_DIR=/encrypted/
    ENCRYPTED_DIR2=/encrypted2/
    TMP_ENC_DIR=/enc_int/
    TMP_DEC_DIR=/dec_int/
    SCRIPTS_DIR=/scripts/
    CSP_DIR=/csp/

    Enc_Image_Setup () {

# This line creates a docker container (with ubuntu as the operating system), running in the background (-d),
# with a pseudo terminal attached (-t), and the capability to receive input from the keyboard (-i)

    docker run -d -i -t --name="ENC_IMAGE" ubuntu

# The following line runs a shell in the created docker container, and executes commands to:
# - create the temp directories used during the encryption/decryption process, as well as the directory for holding
# the scripts
# - update the list of Ubuntu repository packages ("apt-get update"), which is used when installing packages from
# the Ubuntu repositories via the apt-get install <package> command. The command "apt-get update" doesn't install
# anything in the container. Here, we are trusting that when this command is executed, a connection is established to
# the Ubuntu repositories, and the list of packages, with the correct signature for each package, is retrieved from
# the repository.
# - install the build-essential package from Ubuntu's repository # ("apt-get -y install build-essential"), that is needed for
# building applications from source. We need this package to build the encryption application. This package consists
# of references to packages needed for building software on Ubuntu, including compilers (gcc and g++), GNU C library,
# and the MAKE utility.
# NOTE: We assume that these packages are not malicious, since they are obtained from the Ubuntu repository, and
# we are trusting our operating system, and by extension the OS provider. However, even if any of these packages is
# malicious, and it colludes with the CSP client (under this configuration, we assume some malicious components may
# collude), it would only be able to leak the user's data, or cryptographic keys via some covert channel that isn't
# addressed by our design. Because, though these packages are installed in each of the encryption containers, and thus
# would be able to access the user's unencrypted data, as well as cryptographic keys, they are not installed in the
# Verification container.
```

```
# - install the incron filesystem monitoring utility ("apt-get install incron")
# NOTE: This is a third party application, that has access to the user's unencrypted files, and also may be able to access
# the user's crypto keys. Under this configuration, we assume some malicious components may collude, therefore if this
# filesystem monitoring tool is malicious, and colludes with the CSP client application, it would be able to leak the
# user's data, or cryptographic keys. We assume that this application is trusted not to collude with the CSP client
# application.
# - enable the "root" user to use incron within the container ("echo 'root' > /etc/incron.allow")
# - create the filesystem monitoring rules for incron, and save them (in /var/spool/incron/root)
# The rules specify which directory to monitor (e.g. $DECRYPTED_DIR), which event to monitor for
# (e.g. IN_CLOSE_WRITE), and what action to take
# (e.g. $SCRIPTS_DIR/encrypt_script.pl $% $'@'$'#' $ENCRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR
# $SCRIPTS_DIR/keyfile' – this calls the "encrypt_script" (which we provide, and the user can audit), and passes it the
# name of the event {$%} that triggered incron, the url {$@$#} of the file that triggered incron, the names of the
# directories used during the encryption/decryption process, as well as the url of the keyfile containing the encryption
# key/password).
# For this configuration, we require the two encryption applications to use the same key and initialization
# vector when encrypting the user's files, so that we can verify that the resulting ciphertext is the same. We therefore
# generate the key and IV during this stage, and store the values in the file "keyfile". This is achieved
# with the following commands:
# dd if=/dev/urandom of=$SCRIPTS_DIR/keyfile.data bs=1024 count=100;
# md5sum $SCRIPTS_DIR/keyfile.data | cut -d ' ' -f1 > $SCRIPTS_DIR/keyfile
#
# The value stored in the "keyfile" is a 32 byte string, from which a 16 byte # key and a 16 byte IV are extracted, by
# the scripts (encrypt_script.pl and decrypt_script.pl) in the encryption containers, and then passed to the encryption
# application during encryption/decryption. A new value is generated randomly each time this configuration script is
# run, and the user can also edit the keyfile, to manually change/update this value. Our goal here is to generate a default
# key and IV, that are hard to guesss and that can be used by the user for encryption/decryption of files, as well as give
the
# user the flexibility to change the key/IV values whenever he/she wants to.
#
# So when the Enc_Mod_Setup() function is called to create the encryption containers from this image, they will
# both have the same keyfile copy in the SCRIPTS_DIR. In order to share files with multiple devices, each running this
# configuration, the user will need to securely transfer one of the "keyfile" copies from an encryption container, to the
# encryption containers created on the other devices.

    docker exec ENC_IMAGE sh -c "mkdir $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR; \
    apt-get update; apt-get -y install build-essential; apt-get install incron; \
    echo 'root' > /etc/incron.allow; \

    echo '$DECRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/encrypt_script.pl $% $'@'$'#' \
    $ENCRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root; \

    echo '$ENCRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/decrypt_script.pl $% $'@'$'#' \
    $DECRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root; \

    dd if=/dev/urandom of=$SCRIPTS_DIR/keyfile.data bs=1024 count=100; \
    md5sum $SCRIPTS_DIR/keyfile.data | cut -d ' ' -f1 > $SCRIPTS_DIR/keyfile"

# The following three lines, stop the ENC_IMAGE container, save its image, and then delete the container.
# NOTE: We required network access to download and install the components to build the encryption application,
# however our security requirements are that the encryption container should have no network access. We therefore
# delete this container which was built with network access, and then later (in the Enc_Mod_Setup function) use the
# saved image (which has the libraries/tools require to build the encryption application) to create an encryption container
# that has no network access.

    docker stop ENC_IMAGE
    docker commit ENC_IMAGE $1
    docker rm ENC_IMAGE
```

```
114)      }
115)
116)      Aux_Image_Setup () {
117)
118)  # This function builds, and saves the image from which we create the INIT_MODULE, and
119)  # VERIFICATION_MODULE containers, which have a similar structure.
120)
121)      docker run -d -i -t --name="AUX_IMAGE" ubuntu
122)      docker exec AUX_IMAGE sh -c "mkdir $SCRIPTS_DIR; \
123)      apt-get update; apt-get -y install build-essential; apt-get install incron; \
124)      echo 'root' > /etc/incron.allow; \
125)
126)      echo '$DECRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/in-out_transfer.pl $% $'@'$'#' \
127)      $ENCRYPTED_DIR $ENCRYPTED_DIR2' >> /var/spool/incron/root; \
128)
129)      echo '$ENCRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/out-in_transfer.pl $% $'@'$'#' \
130)      $ENCRYPTED_DIR2 $DECRYPTED_DIR' >> /var/spool/incron/root; \
131)
132)      echo '$ENCRYPTED_DIR2 IN_CLOSE_WRITE $SCRIPTS_DIR/out-in_transfer2.pl $% $'@'$'#' \
133)      $ENCRYPTED_DIR $DECRYPTED_DIR' >> /var/spool/incron/root"
134)
135)  # The following three lines, stop the AUX_IMAGE container, save its image, and then delete the container.
136)  # NOTE: We required network access to setup the incron utility, however, there is no other reason for the
137)  # INIT_MODULE and VERIFICATION_MODULE containers to have networks access. We therefore
138)  # delete this container which was built with network access, and then later (in the Aux_Mod_Setup function)
139)  # use the saved image to create the container we need, with network access disabled.
140)
141)      docker stop AUX_IMAGE
142)      docker commit AUX_IMAGE $1
143)      docker rm AUX_IMAGE
144)      }
145)
146)      Enc_Mod_Setup () {
147)
148)  # This line creates the encryption container, from the previously saved image, with the two interface directories,
149)  # specified by the user, mounted in the container (-v), and with no network access (–net=none).
150)
151)      docker run -d -i -t --net=none -v $1:$DECRYPTED_DIR -v $2:$ENCRYPTED_DIR --name=$3 $4
152)      }
153)
154)      CSP_Mod_Setup () {
155)
156)  # This line creates the CSP client container, with network access enabled, so that the CSP client can communicate
157)  # with the CSP servers.
158)
159)      docker run -d -i -t -v $1:$CSP_DIR --name=$2 ubuntu
160)      docker exec $2 apt-get update
161)      }
162)
163)      Aux_Mod_Setup () {
164)
165)  # This function is used to create the INIT and VERIFICATION containers, with network access disabled
166)
167)       docker run -d -i -t -net=none -v $1:$DECRYPTED_DIR -v $2:$ENCRYPTED_DIR -v $3:$ENCRYPTED_DIR2
      --name=$4 $5
168)      }
169)
170)  # We start by creating the "Base" directory, and the six shared directories.
```

```
171)             mkdir ${BASE_DIR} ${USER_DIR} ${INIT_ENC1_INTERFACE} ${INIT_ENC2_INTERFACE}
    ${ENC1_VER_INTERFACE} ${ENC2_VER_INTERFACE} ${VER_CSP_INTERFACE}
172)
173) # We create, and save, a template for the Encryption application containers to be built.
174)     Enc_Image_Setup ${ENC_IMAGE}
175)
176) # We create, and save, a template for the INIT and VERFIFCATION containers to be built.
177)     Aux_Image_Setup ${AUX_IMAGE}
178)
179) # We build the INIT_MODULE container, with network acces disabled. The Aux_Mod_Setup function
180) # takes in 5 parameters:
181) # - the names of the 3 interface directories through which the INIT container interfaces with
182) # the user (USER_DIR), and with the two encryption containers (INIT_ENC1_INTERFACE and
183) # INIT_ENC2_INTERFACE)
184) # - the name of the container
185) # - the name of the image used to build the INIT container
186)     Aux_Mod_Setup ${USER_DIR} ${INIT_ENC1_INTERFACE} ${INIT_ENC2_INTERFACE} ${INIT_MODULE}
    ${AUX_IMAGE}
187)
188) # We build the two Encryption containers, with no network access. The Enc_Mod_Setup function takes
189) # in 4 parameters:
190) # - the names of the 2 interface directories through which the encryption container receives data to be
191) # encrypted, and delivers encrypted data.
192) # - the name of the container
193) # - the name of the image used to build the encryption container
194)     Enc_Mod_Setup ${INIT_ENC1_INTERFACE} ${ENC1_VER_INTERFACE} ${ENC_MODULE} ${ENC_IMAGE}
195)             Enc_Mod_Setup ${INIT_ENC2_INTERFACE} ${ENC2_VER_INTERFACE} ${ENC_MODULE2}
    ${ENC_IMAGE}
196)
197) # We build the VERIFICATION_MODULE container, also with network access disabled.
198) # The Aux_Mod_Setup function takes in 5 parameters:
199) # - the names of the 3 interface directories through which the container interfaces with the CSP client
200) # container (VER_CSP_INTERFACE), and with the two encryption containers (ENC1_VER_INTERFACE
201) # and ENC2_VER_INTERFACE)
202) # - the name of the container
203) # - the name of the image used to build the VERFICATION container
204)     Aux_Mod_Setup ${VER_CSP_INTERFACE} ${ENC1_VER_INTERFACE} ${ENC2_VER_INTERFACE} ${VER-
    IFICATION_MODULE} ${AUX_IMAGE}
205)
206) # Finally, we build the CSP module. The CSP_Mod_Setup function takes in 2 parameters:
207) # - the name of the interface directory through which the CSP client container receives data
208) # from, or to be synced to, the CSP
209) # - the name of the CSP client container
210)     CSP_Mod_Setup ${VER_CSP_INTERFACE} ${CSP_MODULE}
```

*F. Counter-CovertChannels Configuration Encryption Script (encrypt_script.pl)*

```
1  #! /usr/bin/perl
2
3  use warnings;
4  use strict;
5
6  use File::Compare;
7  use File::Basename;
8  use File::Copy;
9
10 # The following variables are initialised using values passed from
11 # the configuration setup file (counter-covertchannels_config.sh)
12 # - The variable "url" is assigned the full path of the file that
13 # triggered this script.
14 # - The variable "encrypted" is assigned the full path of the directory
15 # where encrypted files are stored
16 # - The "tmpEnc" and "tmpDec" directories are assigned the full
17 # paths of the directories used to temporarily hold files during the
18 # encryption/decryption process.
19 # - The "keyFile" variable is assigned the full path of the file
20 # containing the key, and initialisation vector (IV), used for encryption.
21
22 my $url = $ARGV[1];
23 my $encrypted = $ARGV[2];
24 my $tmpEnc = $ARGV[3];
25 my $tmpDec = $ARGV[4];
26 my $keyFile = $ARGV[5];
27 my $key = "";
28 my $iv = "";
29
30 sub Encrypt{
31
32 # This function should be modified, to suit the requirements
33 # of the encryption application being used. It receives the
34 # url of the file to be encrypted, its name, and the encryption
35 # key and IV. We then specify the name(path + extension) of
36 # the output file to be created after encryption, call the
37 # encryption application, and return the full path to the
38 # encrypted file.
39 # Note that the encrypted file is placed in the tmpEnc
40 # directory. It is later (after this function returns) copied to
41 # the encrypted directory.
42
43         my $fileURL = $_[0];
44         my $fileName = $_[1];
45         my $encKey = $_[2];
46         my $encIV = $_[3];
47         my $ext = '.enc';
48         my $file = "$tmpEnc"."$fileName";
49         my $output = "$tmpEnc"."$fileName"."$ext";
50
51         copy($fileURL, $tmpEnc);
52         `sisicrypt -e -f $file -k $encKey -i $encIV`;
53         unlink $file;
54         return $output;
55 }
56
```

```
57 # This script monitors the "decrypted" directory via which, plaintext
58 # files are exchanged with the INIT container. When a file, to be
59 # encrypted (received from the INIT container), or  which has just been
60 # decrypted (to be sent to the INIT container),  lands on this directory,
61 # this script is triggered, and the url of the file is passed to this
62 # script.
63
64 chomp($url);
65
66 # we extract the file name from the url
67 my $fname = fileparse($url);
68 my $compareFile = $tmpDec . $fname;
69
70 # We check if the file that triggered the script has a similar copy
71 # in the "tmpDec" directory. If a similar copy exists in the tmpDec
72 # directory, it means that the file should not be encrypted, since
73 # it has just been decrypted.
74 # When the "decrypt_script", in charge of decryption, decrypts a file,
75 # it places a copy of the decrypted plaintext file in the tmpDec
76 # directory, before copying the file to the "decrypted" directory.
77 # This helps distinguish between the case where a file entering the
78 # "decrypted" directory(which triggers this script), is from the CSP
79 # and has just been decrypted, or is from the user and needs to be
80 # encrypted before being  sent to the CSP.
81
82 my $fileCheck = compare($url, $compareFile);
83
84 if ($fileCheck == 0){
85
86         # if a similar copy of the file exists in the tmpDec directory,
87         # we delete the copy of the file in the tmpDec directory, and exit.
88         # There's nothing else to do.
89         unlink $compareFile;
90 } else{
91
92         # otherwise, we open the file containing the encryption key and IV,
93         # extract the two values, and close the file.
94
95         open(KEYFILE, "<$keyFile") || die("failed to open $keyFile\n");
96         if ((read KEYFILE, $key, 16) != 16) {die("failed to read key\n")};
97         if ((read KEYFILE, $iv, 16) != 16) {die("failed to read iv\n")};
98         close(KEYFILE);
99
100        # we then pass the url of the file to be encrypted, the filename, and
101        # the encryption key and IV, to the encryption function.
102        # The encryption function encrypts the file, then returns the full path
103        # to the encrypted file.
104        # The encrypted file is then copied to the encrypted directory, and this
105        # script's work is done.
106
107        my $outFile = Encrypt($url, $fname, $key, $iv);
108        copy($outFile, $encrypted);
109 }
```

*G. Counter-CovertChannels Configuration Decryption Script (decrypt_script.pl)*

```
#! /usr/bin/perl

use warnings;
use strict;

use File::Compare;
use File::Basename;
use File::Copy;

# The following variables are initialised using values passed from
# the configuration setup file (counter-covertchannels_config.sh)
# - The variable "url" is assigned the full path of the file
# that triggered # this script.
# - The variable "decrypted" is assigned the full path of the directory
# where decrypted files are stored # - The "tmpEnc" and "tmpDec"
# directories are assigned the full paths of the directories used to
# temporarily hold files during the encryption/decryption process.
# - The "keyFile" variable is assigned the full path of the file
# containing the key and initialisation vector (IV) used for decryption

my $url = $ARGV[1];
my $decrypted = $ARGV[2];
my $tmpEnc = $ARGV[3];
my $tmpDec = $ARGV[4];
my $keyFile = $ARGV[5];
my $key = "";
my $iv = "";

sub Decrypt{

# This function should be modified, to suit the requirements
# of the encryption application being used. It receives the
# url of the file to be decrypted, its name, and the decryption
# key and IV. We then specify the name(path) of the output
# file to be created after decryption, call the encryption
# application, and return the full path to the decrypted file.
# Note that the decrypted file is placed in the tmpDec
# directory. It is later (after this function returns) copied to
# the "decrypted" directory.

        my $fileURL = $_[0];
        my $fileName = $_[1];
        my $encKey = $_[2];
        my $encIV = $_[3];
        my $file = "$tmpDec"."$fileName";
        my $output = "$tmpDec"."$fileName";
        $output =~ s/.enc//;

        copy($fileURL, $tmpDec);
        `sisicrypt -d -f $file -k $encKey -i $encIV`;
        unlink $file;
        return $output;
}

# This script monitors the "encrypted" directory which holds
# ciphertext files. When an encrypted file, received from, or
```

```
57 # being sent to, the  VERFICATION container, lands in this directory,
58 # this script is triggered, and the url of the file is passed to this script.
59
60 chomp($url);
61
62 # we extract the file name from the url
63 my $fname = fileparse($url);
64 my $compareFile = $tmpEnc . $fname;
65
66 # We check if the file that triggered the script has a similar copy
67 # in the tmpEnc directory. If a similar copy exists in the tmpEnc
68 # directory, it means that the file should not be decrypted, since
69 # it has just been encrypted. When the "encrypt_script", in charge of
70 # encryption, encrypts a file, it places a copy of the encrypted file
71 # in the tmpEnc directory, before copying the file to the "encrypted"
72 # directory. This helps distinguish between the case where a
73 # file entering the  "encrypted" directory(which triggers this script),
74 # is from the CSP and  needs to be decrypted, or is from the user and
75 # has just been encrypted.
76
77 my $fileCheck = compare($url, $compareFile);
78
79 if ($fileCheck == 0){
80
81         # if a similar copy of the file exists in the tmpEnc directory,
82         # we delete the copy of the file in the tmpEnc directory, and exit.
83         # There's nothing else to do.
84
85         unlink $compareFile;
86 } else{
87
88         # otherwise, we open the file containing the decryption key and IV,
89         # extract the two values, and close the file.
90
91         open(KEYFILE, "<$keyFile") || die("failed to open $keyFile\n");
92         if ((read KEYFILE, $key, 16) != 16) {die("failed to read key\n")};
93         if ((read KEYFILE, $iv, 16) != 16) {die("failed to read iv\n")};
94         close(KEYFILE);
95
96         # we then pass the url of the file to be decrypted, the filename, and
97         # the decryption key and IV, to the Decrypt function. The Decrypt
98         # function decrypts the file, then returns the full path to the decrypted
99         # file. The decrypted file is then copied to the "decrypted" directory,
100        # and this script's work is done.
101
102        my $outFile = Decrypt($url, $fname, $key, $iv);
103        copy($outFile, $decrypted);
104 }
```

*H. Counter-CovertChannels Configuration Encryption Wrapper code (sisicrypt.h)*

```
1  #ifndef SISICRYPT_H
2          #include <stdio.h>
3          #include <stdlib.h>
4          #include <unistd.h>
5          #include <string.h>
6          #include <path to crytpo library header file>
7          #define SISICRYPT_H
8
9          typedef struct{
10                 int encrypt;            /* -e option */
11                 int decrypt;            /* -d option */
12                 char *inFileName;       /* -f option */
13                 char *key;                      /* -k option */
14                 char *iv;                       /* -i option */
15         } cmdArgs;
16
17         cmdArgs args;
18         static const char *optString = "edf:k:i:h";
19         int BUFFERLENGTH = 16;
20         void Usage(void);
21         void ProcessCmdLine(int argc, char *argv[]);
22         void EncryptFile(char *fileName, char *key, char *iv);
23         void DecryptFile(char *fileName, char *key, char *iv);
24         void Error(char * errorMessage);
25 #endif
```

*I. Counter-CovertChannels Configuration Encryption Wrapper code (sisicrypt.c)*

```
1  #include "sisicrypt.h"
2  int main(int argc, char *argv[]){
3          /* initialize cmdArgs */
4          args.encrypt = 0;
5          args.decrypt = 0;
6          args.inFileName = NULL;
7          args.key = NULL;
8          args.iv = NULL;
9
10         ProcessCmdLine(argc, argv);
11
12         if (args.encrypt == 0 && args.decrypt == 0) {
13         Usage();
14         } else if (args.encrypt == 1){
15                 EncryptFile(args.inFileName, args.key, args.iv);
16         } else{
17                 DecryptFile(args.inFileName, args.key, args.iv);
18         }
19         exit(EXIT_SUCCESS);
20 }
21
22 void ProcessCmdLine(int argc, char *argv[]) {
23         if (argc < 8 || argc > 8) Usage();
24         int opt = 0;
25         opt = getopt(argc, argv, optString);
26         while (opt != -1){
27                 switch(opt) {
28                                 case 'e':
29                                         if (args.decrypt == 1) Usage();
30                                         args.encrypt = 1;
31                                         break;
32                                 case 'd':
33                                         if (args.encrypt == 1) Usage();
34                                         args.decrypt = 1;
35                                         break;
36                                 case 'f':
37                                         args.inFileName = optarg;
38                                         break;
39                                 case 'k':
40                                         args.key = optarg;
41                                         break;
42                                 case 'i':
43                                         args.iv = optarg;
44                                         break;
45                                 case 'h':
46                                         Usage();
47                                         break;
48                                 default:
49                                         break;
50                 }
51         opt = getopt(argc, argv, optString);
52         }
53 }
54
55 void EncryptFile( char *fileName, char *key, char *iv ) {
56         int status;
```

```
57          char  buffer[BUFFERLENGTH];
58          char *outFile;
59          FILE *inFp = NULL, *outFp = NULL;
60
61          outFile = calloc((strlen(fileName) + 4 + 1), 1);
62          if (outFile == NULL){
63                  Error("Failed to allocate memory for outFile!\n");
64          }
65          strcpy(outFile, fileName);      strcat(outFile, ".enc");
66
67          outFp = fopen(outFile, "w");
68          if (outFp == NULL){
69                  Error("failed to open output file for writing.");
70          }
71          inFp = fopen(fileName, "r");
72          if (inFp == NULL) {
73                  Error("failed to open input file for reading!");
74          }
75          memset(buffer, 0, BUFFERLENGTH);
76
77          /* functions to initialise crypto library and setup encryption context */
78
79          while (fread(buffer, 1, 16, inFp) > 0) {
80                  // function(s) to encrypt data in buffer
81
82                  if(fwrite(buffer, 1, 16, outFp) != BUFFERLENGTH) {
83                          Error("There was an error in writing encrypted output to
                                outFile!");
84                  }
85                  memset(buffer, 0, BUFFERLENGTH);        }
86                  if (ferror(inFp)){
87                          Error("I/O error: Didn't complete reading from input file!\n"
                                );
88                  }
89
90          /* function to destroy encryption context, and shutdown crypto library */
91
92          fclose(inFp);
93          fclose(outFp);
94          free(outFile);
95  }
96
97  void DecryptFile(char *fileName, char *key, char *iv) {
98          int status;
99          char  buffer[BUFFERLENGTH];
100         char *outFile;
101         FILE *inFp = NULL, *outFp = NULL;
102
103         outFile = calloc((strlen(fileName) + 1 - 4), 1);
104         if (outFile == NULL) {
105                 Error("Failed to allocate memory for outFile!\n");
106         }
107         strncpy(outFile, fileName, (strlen(fileName) - 4));
108
109         outFp = fopen(outFile, "w");
110         if (outFp == NULL) {
111                 Error("failed to open output file for writing.");
112         }
```

```
113
114         inFp = fopen(fileName, "r");
115         if (inFp == NULL) {
116                 Error("failed to open input file for reading!");
117         }
118         memset(buffer, 0, BUFFERLENGTH);
119
120         /* functions to initialise crypto library and setup encryption context */
121
122         while (fread(buffer, 1, 16, inFp) > 0) {
123
124                 // function(s) to decrypt data in buffer
125
126                 if(fwrite(buffer, 1, 16, outFp) != BUFFERLENGTH){
127                         Error("There was an error in writing decrypted output to
                                outFile!");
128                 }
129                 memset(buffer, 0, BUFFERLENGTH);
130         }
131         if (ferror(inFp)){
132                 Error("I/O error: Didn't complete reading from input file!\n");
133         }
134
135         /* function to destroy encryption context, and shutdown crypto library */
136
137         fclose(inFp);
138         fclose(outFp);
139         free(outFile);
140 }
141
142 void Usage(void) {
143         printf("Usage:\nTo Encrypt file:\t sisicrypt -e -f <filename> -k <encryption
            key> -i <initialisation vector>\n");
144         printf("To Decrypt file:\t sisicrypt -d -f <filename> -k <encryption key> -i
            <initialisation vector>\n");       exit(EXIT_FAILURE);
145 }
146
147 void Error(char * errorMessage){
148         fprintf(stderr, "%s\n", errorMessage);
149         exit(EXIT_FAILURE);
150 }
```

*J. Counter-CovertChannels Configuration Init Module Script (in-out_transfer.pl)*

```
1  #! /usr/bin/perl
2
3  use warnings;
4  use strict;
5
6  use File::Compare;
7  use File::Basename;
8  use File::Copy;
9
10 # The following variables are initialised using values passed from
11 # the configuration setup file (counter-covertchannels_config.sh)
12 # - The variable "url" is assigned the full path of the file that
13 # triggered this script.
14 # - The variable "encrypted" is assigned the full path of the
15 # directory that interfaces with the ENC_MODULE container
16 # - The variable "encrypted2" is assigned the full path of the
17 # directory that interfaces with the ENC_MODULE2 container
18
19 my $url = $ARGV[1];
20 my $encrypted = $ARGV[2];
21 my $encrypted2 = $ARGV[3];
22
23 # This script monitors the "decrypted" directory in the INIT  container,
24 # which interfaces with the User's directory. When a file, to be synced
25 # to the CSP (created, or updated by the user), or which has just been
26 # received from the CSP, lands on this directory, this script is
27 # triggered, by the matching rule specified in the incrontab file
28 # (/var/spool/incron/root),  and this matching rule passes the url of
29 # the file to this script.
30 # Note that the rules, specifiying the directories to be monitored by
31 # incron, the events to monitor, and the actions to take when the events
32 # occur, are created during the configuration setup phase
33 # (counter-covertchannels_config.sh -- specifically in the Aux_Image_Setup()
34 # function)
35
36 chomp($url);
37
38 # we extract the file name from the url
39
40 my $fname = fileparse($url);
41 my $compareFile = $encrypted . $fname;
42
43 # We check if the file that triggered the script has a similar copy
44 # in the "encrypted" directory. If a similar copy exists, it means
45 # that the file should not be sent to the two encryption containers
46 # (ENC_MODULE and ENC_MODULE2), since it has just been received from
47 # the "encrypted" directory, and is present in both "encrypted" and
48 # "encrypted2" directories. (We know that it is present in both
49 # direcories, just by checking one directory, since the mechanism that
50 # copies the encrypted file from the "encrypted" directory to the
51 # "decrypted" directory, only does so when it finds an exact copy
52 # of the file in the "encrypted2" directory within a specified
53 # time constraint. Refer to the out-in_transfer.pl script to see
54 # how this works.)
55 # This helps distinguish between the case where a file entering the
56 # "decrypted" directory (which triggers this script), is from the CSP
```

```
57 # and has just been decrypted, or is from the user and needs to be
58 # encrypted before being sent to the CSP. We need to distinguish these
59 # two cases to avoid getting into an infinite loop.
60
61 my $fileCheck = compare($url, $compareFile);
62
63 if ($fileCheck != 0){
64
65 # If a similar copy isn't found in the "encrypted" directory, it
66 # means that the file is to be encrypted, then sent to the CSP.
67 # We thus copy the file to the directories that interface with the
68 # two encryption containers(ENC_MODULE and ENC_MODULE2).
69 # The objective is to have the same file encrypted by the two
70 # encryption applications in ENC_MODULE and ENC_MODULE2
71 # respectively, using the same key and initialization vector, so
72 # that we can verify whether they are both producing the same output
73
74         copy($url, $encrypted);
75         copy($url, $encrypted2);
76 }
```

*K. Counter-CovertChannels Configuration Init Module Script (out-in_transfer.pl)*

```
1  #! /usr/bin/perl
2
3  use warnings;
4  use strict;
5
6  use File::Compare;
7  use File::Basename;
8  use File::Copy;
9
10 # The following variables are initialised using values passed from
11 # the configuration setup file (counter-covertchannels_config.sh)
12 # - The variable "url" is assigned the full path of the file
13 # that triggered this script.
14 # - The variable "encrypted2" is assigned the full path of the
15 # directory that interfaces with the ENC_MODULE2 container
16 # - The variable "plainDir" is assigned the full path of the
17 # directory # that interfaces with the user's directory(USER_DIR)
18 # - The "auditLog" variable is assigned the full path of the file
19 # we use to log events that may be malicious.
20
21 my $url = $ARGV[1];
22 my $encrypted2 = $ARGV[2];
23 my $plainDir = $ARGV[3];
24 my $auditLog = "/init-mod_audit.log";
25
26 sub FileVerification{
27
28 # This function takes in 3 input parameters, the URL of the file
29 # that triggered this script, the name of the file, and the path
30 # to the "encrypted2" directory.
31 # It then checks whether a similar (content) copy of the file that
32 # triggered this script, exists in the "encrypted2" directory.
33 # If it finds one, it copies the file that triggered this script to
34 # the plainDir directory.
35 # It returns a status code indicating whether or not a similar file
36 # was found.
37
38         my $fileURL = $_[0];
39         my $fileName = $_[1];
40         my $dir = $_[2];
41         my $compFile = $dir . $fileName;
42         my $compareStatus = compare($fileURL, $compFile);
43
44         if ($compareStatus == 0){
45                 copy($url, $plainDir);
46         }
47         return $compareStatus;
48 }
49
50 # This script monitors the "encrypted" directory in the INIT
51 # container, which interfaces with the encryption container
52 # ENC_MODULE.
53 # When a file, to be synced to the CSP (created, or updated
54 # by the user), or which has just been received from the CSP,
55 # lands on this directory, this script is triggered, by the
56 # matching rule specified in the incrontab file
```

```
57 # (/var/spool/incron/root), and this matching rule passes the
58 # url of the file to this script.
59 # Note that the rules, specifiying the directories to be
60 # monitored by incron, the events to monitor, and the actions
61 # to take when the events occur, are created during
62 # the configuration setup phase (counter-covertchannels_config.sh
63 # -- specifically in the Aux_Image_Setup() function)
64
65 chomp($url);
66
67 # we extract the file name from the url
68 my $fname = fileparse($url);
69 my $compareFile = $plainDir . $fname;
70
71 # We check if the file that triggered the script has a similar copy
72 # in the plainDir directory. If a similar copy exists, it means that
73 # the file should not be sent to the user's directory, since it has
74 # just been received from the user, and is being synced to the CSP.
75 # This helps distinguish between the case where a file entering
76 # the "encrypted" directory (which triggers this script), is from the
77 # CSP and has just been decrypted, or is from the user and needs to be
78 # encrypted before being sent to the CSP. We need to distinguish
79 # these two cases in order to avoid getting into an infinite loop.
80
81 my $fileCheck = compare($url, $compareFile);
82
83 if($fileCheck != 0){
84
85 # If a similar copy isn't found in the plainDir directory, it means that
86 # the file has been received from the CSP, and is to be copied to the
87 # user's directory.
88 # We first verify that a similar copy of this file was received in the
89 # "encrypted2" directory (which interfaces with the ENC_MODULE2 container).
90 # This is done to ensure that both encryption containers are operating
91 # correctly, and that neither is modifying the files being received from the
92 # CSP. If we find a similar copy of the file in the "encrypted2" directory
93 # we copy this file to the plainDir directory
94
95         my $verStatus = FileVerification($url, $fname, $encrypted2);
96         if($verStatus != 0){
97
98         # if a similar copy of this file isn't found in the "encrypted2" directory,
99         # we wait for a minute, then check again
100
101                 sleep(60);
102                 $verStatus = FileVerification($url, $fname, $encrypted2);
103                 if($verStatus != 0){
104
105                 # if a similar copy of this file still isn't found in the "encrypted2
                        " directory,
106                 # we log this event, and disable incron, which coordinates
                        synchronization, so
107                 # that no more file synchronization with the CSP will take place,
                        until the user
108                 # restarts incron.
109
110                         open(LOG, ">>$auditLog") || die("failed to open $auditLog\n")
                            ;
```

```
111                         print LOG "File check failed! Second copy of file $fname was
                                not seen in $encrypted2!\n";
112                         close(LOG);
113                         `pkill -f incrond`;
114                     }
115             }
116 }
```

*L. Counter-CovertChannels Configuration Init Module Script (out-in_transfer2.pl)*

```
1  #! /usr/bin/perl
2
3  use warnings;
4  use strict;
5
6  use File::Compare;
7  use File::Basename;
8  use File::Copy;
9
10 # The following variables are initialised using values passed from
11 # the configuration setup file (counter-covertchannels_config.sh)
12 # - The variable "url" is assigned the full path of the file
13 # that triggered # this script.
14 # - The variable "encrypted" is assigned the full path of the
15 # directory that interfaces with the ENC_MODULE container
16 # - The variable "plainDir" is assigned the full path of the
17 # directory that interfaces with the user's directory(USER_DIR)
18 # - The "auditLog" variable is assigned the full path of the file
19 # we use to log events that may be malicious.
20
21 my $url = $ARGV[1];
22 my $encrypted = $ARGV[2];
23 my $plainDir = $ARGV[3];
24 my $auditLog = "/init-mod_audit.log";
25
26 sub FileCheck{
27
28 # This function takes in 3 input parameters, the URL of the file
29 # that triggered this script, the name of the file, and the path
30 # of the directory, that we want to check has a similar copy of
31 # the file.
32 # It then checks whether a similar (content) copy of the file that
33 # triggered this script, exists in the directory.
34 # It returns a status code indicating whether or not a similar file
35 # was found.
36
37         my $fileURL = $_[0];
38         my $fileName = $_[1];
39         my $dir = $_[2];
40         my $compareFile = "$dir"."$fileName";
41         my $fileCheck = compare($fileURL, $compareFile);
42         return $fileCheck;
43 }
44
45 # This script monitors the "encrypted2" directory in the INIT
46 # container, which interfaces with the encryption container ENC_MODULE2.
47 # When a file, to be synced to the CSP (created, or updated by the
48 # user), or which has just been received from the CSP, lands on this
49 # directory, this script is triggered, by the matching rule
50 # specified in the incrontab file (/var/spool/incron/root), and this
51 # matching rule passes the url of the file to this script.
52 # Note that the rules, specifiying the directories to be monitored by incron,
53 # the events to monitor, and the actions to take when the events occur,
54 # are created during the configuration setup phase
55 # (counter-covertchannels_config.sh -- specifically in the Aux_Image_Setup()
56 # function)
```

```
57
58 chomp($url);
59
60 # we extract the file name from the url
61 my $fname = fileparse($url);
62 my $status = FileCheck($url, $fname, $plainDir);
63
64 # We check if the file that triggered the script has a similar copy
65 # in the plainDir directory. If a similar copy exists, it means that
66 # the file should not be sent to the user's directory, since it has
67 # just been received from the user, and is being synced to the CSP.
68 # This helps distinguish between the case where a file entering
69 # the "encrypted2" directory (which triggers this script), is from
70 # the CSP and has just been decrypted, or is from the user and needs
71 # to be encrypted before being sent to the CSP. We need to distinguish
72 # these two cases in order to avoid getting into an infinite loop.
73
74 if($status != 0){
75
76         # If a similar copy isn't found in the plainDir directory,
77     # it means that the file has been received from the CSP.
78     # We verify that a similar copy of this file was received in the
79     # "encrypted" directory (which interfaces with the ENC_MODULE
80     # container).
81         # This is done to ensure that both encryption containers are
82         # operating correctly, and that neither is modifying the files
83         # being received from the CSP.
84     # If we find a similar copy of the file in the "encrypted" directory,
85     # we do nothing and exit.
86
87         my $verStatus = FileCheck($url, $fname, $encrypted);
88         if($verStatus != 0){
89
90                 # if a similar copy of this file isn't found in the "encrypted"
91                 # directory, we wait for a minute, then check again
92
93                 sleep(60);
94                 $verStatus = FileCheck($url, $fname, $encrypted);
95
96                 if($verStatus != 0){
97
98                         # if a similar copy of this file still isn't found in the "
                            encrypted"
99                         # directory, we log this event, and disable incron, which
100                        # coordinates synchronization, so that no more file
                            synchronization
101                        # with the CSP will take place, until the user restarts
                            incron.
102                        # we also delete the file that triggered this script, so
                            that
103                        # later on if a similar file arrives in the "encrypted"
                            directory,
104                        # it won't get copied to the user's directory.
105
106                        unlink $url;
107                        open(LOG, ">>$auditLog") || die("failed to open $auditLog\n")
                            ;
108                        print LOG "File check failed! Second copy of file $fname was
```

```
                    not seen in $encrypted!\n";
109                 close(LOG);
110                 `pkill -f incrond`;
111             }
112         }
113 }
```

*M. Counter-CovertChannels Configuration Verification Module Script (in-out_transfer.pl)*

```
1  #! /usr/bin/perl
2
3  use warnings;
4  use strict;
5
6  use File::Compare;
7  use File::Basename;
8  use File::Copy;
9
10 # The following variables are initialised using values passed from
11 # the configuration setup file (counter-covertchannels_config.sh)
12 # - The variable "url" is assigned the full path of the file that
13 # triggered this script.
14 # - The variable "encrypted" is assigned the full path of the
15 # directory that interfaces with the ENC_MODULE container
16 # - The variable "encrypted2" is assigned the full path of the
17 # directory that interfaces with the ENC_MODULE2 container.
18
19 my $url = $ARGV[1];
20 my $encrypted = $ARGV[2];
21 my $encrypted2 = $ARGV[3];
22
23 # This script monitors the "decrypted" directory in the VERIFICATION
24 # container, which interfaces with the CSP's directory. When a file,
25 # to be synced to the CSP (created, or updated by the user), or which
26 # has just been received from the CSP, lands on this directory, this
27 # script is triggered, by the matching rule specified in the incrontab
28 # file (/var/spool/incron/root), and this matching rule passes the
29 # url of the file, as well as the directory arguments, to this
30 # script.
31 # Note that the rules, specifiying the directories to be monitored by
32 # incron, the events to monitor, and the actions to take when the events
33 # occur, are created during the configuration setup phase
34 # (counter-covertchannels_config.sh -- specifically in the Aux_Image_Setup()
35 # function)
36
37 chomp($url);
38
39 # we extract the file name from the url
40 my $fname = fileparse($url);
41 my $compareFile = $encrypted . $fname;
42
43 # We check if the file that triggered the script has a similar copy
44 # in the "encrypted" directory, which interfaces with the ENC_MODULE
45 # container. If a similar copy exists, it means that the file should
46 # not be sent to the two encryption containers
47 # (ENC_MODULE and ENC_MODULE2), since it has just been received from
48 # the "encrypted" directory, and is present in both "encrypted" and
49 # "encrypted2" directories. (We know that it is present in both
50 # direcories, just by checking one directory, since the mechanism that
51 # copies the encrypted file from the "encrypted" directory to the
52 # "decrypted" directory, only does so when it finds an exact copy
53 # of the file in the "encrypted2" directory within a specified
54 # time constraint. Refer to the out-in_transfer.pl script to see
55 # how this works.)
56 # This helps distinguish between the case where a file entering the
```

```
57 # "decrypted" directory(which triggers this script), is from the CSP
58 # and needs to be decrypted before being sent to the user, or is from
59 # the user. We need to distinguish these two cases to avoid getting into
60 # an infinite loop.
61
62 my $fileCheck = compare($url, $compareFile);
63
64 if ($fileCheck != 0){
65
66 # If a similar copy isn't found in the "encrypted" directory, it means
67 # that the file is to be decrypted, then sent to the user.
68 # We thus copy the file to the directories that interface with
69 # the two encryption containers(ENC_MODULE and ENC_MODULE2)
70
71         copy($url, $encrypted);
72         copy($url, $encrypted2);
73 }
```

*N. Counter-CovertChannels Configuration Verification Module Script (out-in_transfer.pl)*

```
1  #! /usr/bin/perl
2
3  use warnings;
4  use strict;
5
6  use File::Compare;
7  use File::Basename;
8  use File::Copy;
9
10 # The following variables are initialised using values passed from
11 # the configuration setup file (counter-covertchannels_config.sh)
12 # - The variable "url" is assigned the full path of the file
13 # that triggered this script.
14 # - The variable "encrypted" is assigned the full path of the
15 # directory that interfaces with the ENC_MODULE container
16 # - The variable "csp" is assigned the full path of the directory
17 # that interfaces with the CSP client container
18 # - The "auditLog" variable is assigned the full path of the
19 # file we use to log events that may be malicious
20
21 my $url = $ARGV[1];
22 my $encrypted2 = $ARGV[2];
23 my $csp = $ARGV[3];
24 my $auditLog = "/ver-mod_audit.log";
25
26 sub FileVerification {
27
28 # This function takes in 3 input parameters, the URL of the file
29 # that triggered this script, the name of the file, and the path
30 # to the "encrypted2" directory.
31 # It then checks whether a similar (content) copy of the file that
32 # triggered this script, exists in the "encrypted2" directory.
33 # If it finds one, it copies the file that triggered this script to
34 # the "csp" directory.
35 # It returns a status code indicating whether or not a similar file
36 # was found.
37
38          my $fileURL = $_[0];
39          my $fileName = $_[1];
40          my $dir = $_[2];
41          my $compareFile = $dir . $fileName;
42      my $compareStatus = compare($fileURL, $compareFile);
43
44          if ($compareStatus == 0){
45                  copy($fileURL, $csp);
46          }
47          return $compareStatus;
48 }
49
50 # This script monitors the "encrypted" directory in the VERIFICATION
51 # container, which interfaces with the encryption container ENC_MODULE.
52 # When a file, to be synced to the CSP (created, or updated by the
53 # user), or which has just been received from the CSP, lands on this
54 # directory, this script is triggered, by the matching rule
55 # specified in the incrontab file (/var/spool/incron/root), and this
56 # matching rule passes the url of the file to this script.
```

```
57 # Note that the rules, specifiying the directories to be monitored by
58 # incron, the events to monitor, and the actions to take when the events
59 # occur, are created during the configuration setup phase
60 # (counter-covertchannels_config.sh -- specifically in the Aux_Image_Setup()
61 # function)
62
63 chomp($url);
64
65 # we extract the file name from the url
66 my $fname = fileparse($url);
67 my $compareFile = $csp . $fname;
68
69 # We check if the file that triggered the script has a similar copy
70 # in the "csp" directory. If a similar copy exists, it means that the
71 # file should not be sent to the CSP's directory, since it has just
72 # been received from the CSP, and is being synced to the user.
73 # This helps distinguish between the case where a file entering the
74 # "encrypted" directory (which triggers this script), is from the CSP
75 # and needs to be decrypted before being sent to the user, or is from
76 # the user. We need to distinguish these two cases in order to avoid
77 # getting into an infinite loop.
78
79 my $fileCheck = compare($url, $compareFile);
80
81 if($fileCheck != 0){
82
83         # If a similar copy isn't found in the "csp" directory,
84     # it means that the file is being sent to the CSP, and is to
85     # be copied to the "csp" directory.
86         # We first verify that a similar copy of this file was
87         # received in the "encrypted2" directory (which interfaces
88         # with the ENC_MODULE2 container). This is done to ensure that
89         # none of the two encryption applications is attempting to
90         # leak data to the CSP, either by embedding it in the encrypted
91         # file, or via a covert channel where it pads the encrypted file
92         # to a certain size in order to transmit information, or even
93         # sending files that were not sent by the user.
94         # If we find a similar copy of the file in the "encrypted2"
95         # directory, we copy this file to the "csp" directory.
96
97         my $verStatus = FileVerification($url, $fname, $encrypted2);
98         if($verStatus != 0){
99
100                 # if a similar copy of this file isn't found in the "encrypted2"
101                 # directory, we wait for a minute, then check again.
102                 # we do this to force the copying of the file to the CSP, as
103                 # close as possible to the time when it was received by the
104                 # encryption application.
105                 # This prevents a malicious encryption application from being able
106                 # to arbitrarily decide when to send the encrypted file, and thus
107                 # leak data to the colluding CSP by modulating the sending time.
108                 # It also reduces the time window within which a malicious
109                 # encryption application can leak data via a timing channel, to
110                 # 1 minute from when the file is received by the other encryption
111                 # application.
112
113                 sleep(60);
114                 $verStatus = FileVerification($url, $fname, $encrypted2);
```

```
115
116                     if($verStatus != 0) {
117
118                             # if a similar copy of this file still isn't found in
119                             # the "encrypted2" directory, we log this event,
120                             # and disable incron, which coordinates synchronization,
121                             # so that no more file synchronization with the CSP
122                             # will take place, until the user restarts incron
123
124                             open(LOG, ">>$auditLog") || die("failed to open $auditLog\n")
                                    ;
125                             print LOG "File check failed! Second copy of file $fname was
                                    not seen in $encrypted2!\n";
126                             close(LOG);
127                             `pkill -f incrond`;
128                     }
129             }
130     }
```

*O. Counter-CovertChannels Configuration Verification Module Script (out-in_transfer2.pl)*

```
1  #! /usr/bin/perl
2
3  use warnings;
4  use strict;
5
6  use File::Compare;
7  use File::Basename;
8  use File::Copy;
9
10 # The following variables are initialised using values passed from
11 # the configuration setup file (counter-covertchannels_config.sh)
12 # - The variable "url" is assigned the full path of the file
13 # that triggered # this script.
14 # - The variable "encrypted" is assigned the full path of the
15 # directory that interfaces with the ENC_MODULE container
16 # - The variable "csp" is assigned the full path of the
17 # directory that interfaces with the user's directory(USER_DIR)
18 # - The "auditLog" variable is assigned the full path of the
19 # file we use to log events that may be malicious
20
21 my $url = $ARGV[1];
22 my $encrypted = $ARGV[2];
23 my $csp = $ARGV[3];
24 my $auditLog = "/ver-mod_audit.log";
25
26 sub FileCheck {
27
28 # This function takes in 3 input parameters, the URL of the file
29 # that triggered this script, the name of the file, and the path
30 # of the directory, that we want to check has a similar copy of
31 # the file.
32 # It then checks whether a similar (content) copy of the file that
33 # triggered this script, exists in the directory. It returns a
34 # status code indicating whether or not a similar file was found.
35
36         my $fileURL = $_[0];
37         my $fileName = $_[1];
38     my $dir = $_[2];
39         my $compareFile = "$dir"."$fileName";
40         my $fileCheck = compare($fileURL, $compareFile);
41     return $fileCheck;
42 }
43
44 # This script monitors the "encrypted2" directory in the VERIFICATION
45 # container, which interfaces with the encryption container ENC_MODULE2.
46 # When a file, to be synced to the CSP (created, or updated by the
47 # user), or which has just been received from the CSP, lands on this
48 # directory, this script is triggered, by the matching rule
49 # specified in the incrontab file (/var/spool/incron/root), and this
50 # matching rule passes the url of the file to this script.
51 # Note that the rules, specifiying the directories to be monitored by
52 # incron, the events to monitor, and the actions to take when the events
53 # occur, are created during the configuration setup phase
54 # (counter-covertchannels_config.sh -- specifically in the Aux_Image_Setup()
55 # function)
56
```

```
57  chomp($url);
58
59  # we extract the file name from the url
60
61  my $fname = fileparse($url);
62
63  # We check if the file that triggered the script has a similar copy
64  # in the "csp" directory. If a similar copy exists, it means that
65  # that the file should not be sent to the "csp" directory, since it
66  # has just been received from the CSP, and is being synced to the
67  # user. This helps distinguish between the case where a file entering
68  # the "encrypted2" directory(which triggers this script), is from
69  # the CSP and needs to be decrypted before being sent to the user,
70  # or is from the user. We need to distinguish these two cases in
71  # order to avoid getting into an infinite loop.
72
73  my $status = FileCheck($url, $fname, $csp);
74
75
76  if($status != 0){
77
78          # If a similar copy isn't found in the "csp" directory,
79      # it means that the file is being sent to the CSP. We
80          # verify that a similar copy of this file was received
81          # in the "encrypted" directory (which interfaces with
82          # the ENC_MODULE container). This is done to ensure
83          # that none of the two encryption applications is
84          # attempting to leak data to the CSP, either by
85          # embedding it in the encrypted file, or via a covert
86          # channel where it pads the encrypted file to a certain
87          # size in order to transmit information, or even sending
88          # files that were not sent by the user.
89          # If we find a similar copy of the file in the "encrypted"
90          # directory, we do nothing and exit.
91
92          my $verStatus = FileCheck($url, $fname, $encrypted);
93          if($verStatus != 0) {
94
95                  # if a similar copy of this file isn't found in the "encrypted"
96                  # directory, we wait for a minute, then check again. We do this
97                  # to force the copying of the file to the CSP, as close as
98                  # possible to the time when it was received by the encryption
99                  # application.
100                 # This prevents a malicious encryption application from being
101                 # able to arbitrarily decide when to send the encrypted file,
102                 # and thus leak data to the colluding CSP by modulating the
103                 # sending time. It also reduces the time window within which
104                 # a malicious encryption application can leak data via a
105                 # timing channel, to 1 minute from when the file is received by
106                 # the other encryption application.
107
108                 sleep(60);
109                 $verStatus = FileCheck($url, $fname, $encrypted);
110                 if($verStatus != 0){
111
112                         # if a similar copy of this file still isn't found in the
113                         # "encrypted" directory, we log this event, and
114                         # disable incron, which coordinates synchronization, so that
```

```
115                         # no more file synchronization with the CSP will take place,
116                         # until the user restarts incron.
117                         # We also delete the file that triggered this script, so
118                         # that later on if a similar file arrives in the "encrypted"
119                         # directory, it won't get copied to the "csp" directory.
120
121                         unlink $url;
122                         open(LOG, ">>$auditLog") || die("failed to open $auditLog\n")
                                ;
123                         print LOG "File check failed! Second copy of file $fname was
                                not seen in $encrypted!\n";
124                         close(LOG);
125                         `pkill -f incrond`;
126                     }
127             }
128     }
```