

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 party), 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 initial 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

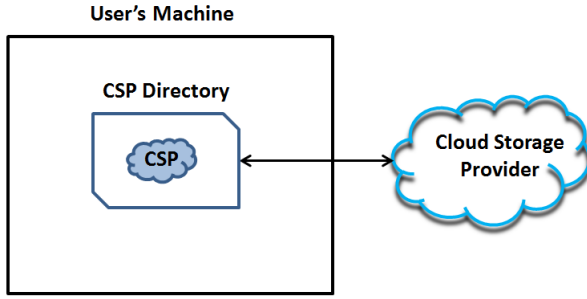


Figure 1. Conventional Configuration

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 confidentiality 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

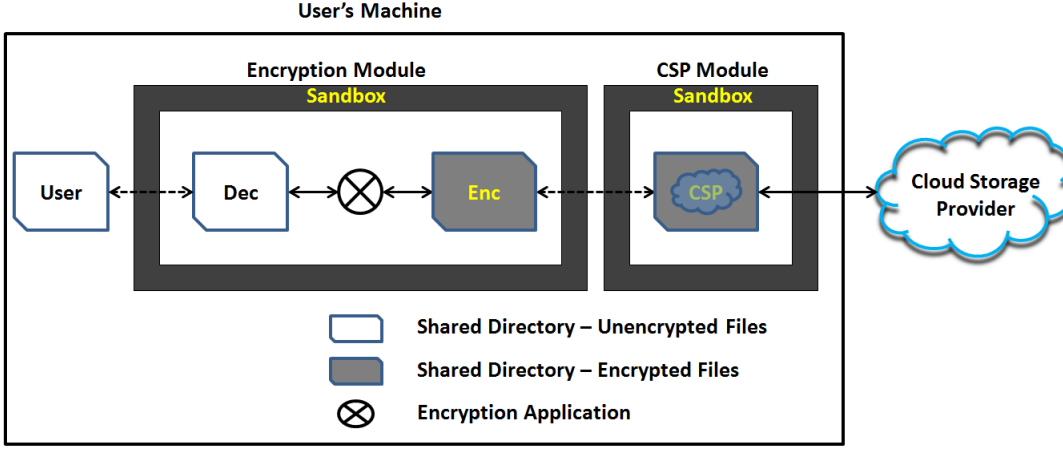


Figure 2. Split-Trust Configuration

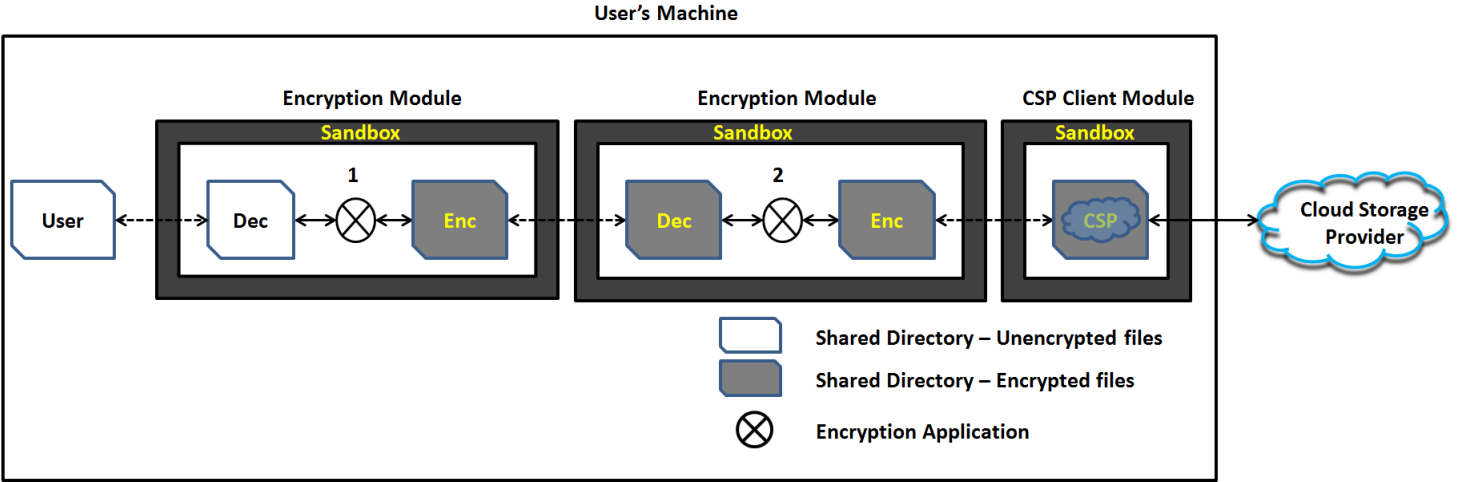


Figure 3. Counter-Collusion Configuration

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

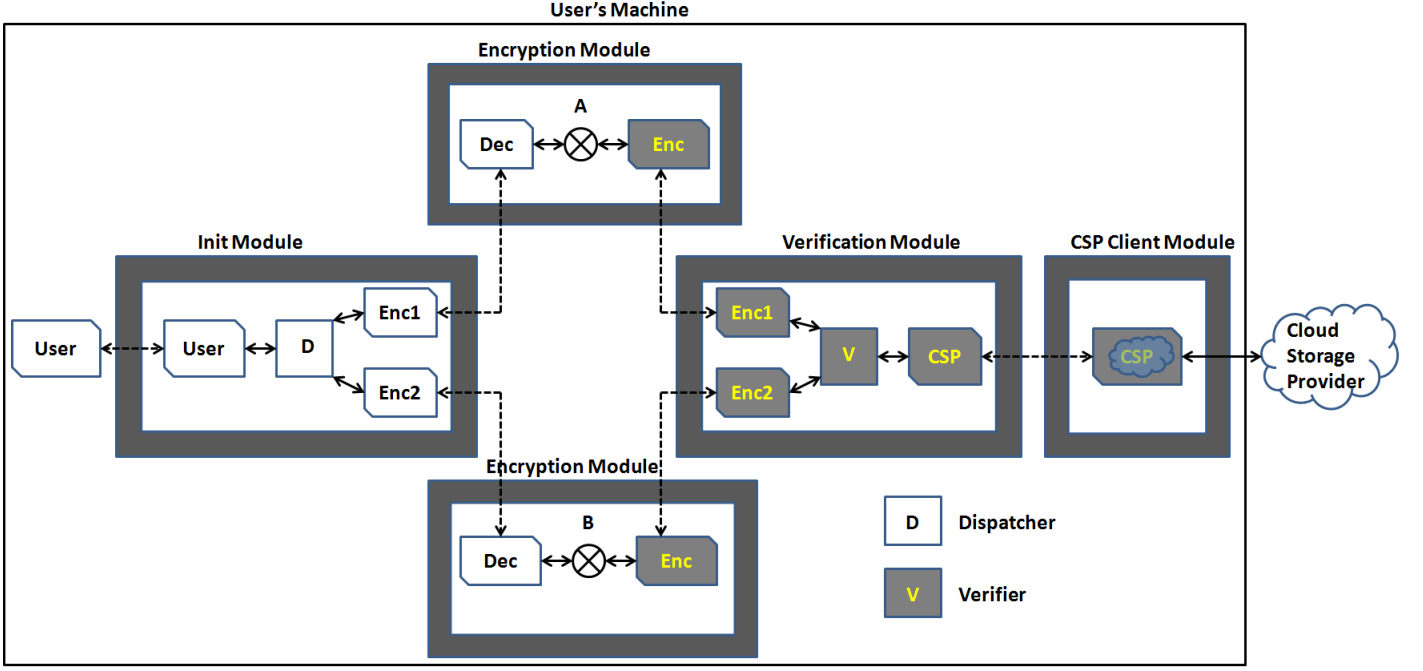


Figure 4. Counter-Covert Channels Configuration

process is triggered by Incon. 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, Incon 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 achieved 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. Encryption application correctly encrypts data. 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 and 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 has access to two encryption applications (from different providers) that achieve this, then there's no need to create custom encryption applications using the provided wrapper code (Appendix H - I).

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 achieved 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 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 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. Our approach ensures that a correctly encrypted copy of the file is sent within a small time window from when the plain text file was received by both encryption applications. It also ensures that, within that window, the encrypted file is sent at a fixed time (top of an even minute). This restricts the malicious encryption application's ability to arbitrarily modulate the sending time to leak information. However, our implementation's accuracy is at a granularity of seconds, meaning the malicious encryption application can still leak information by modulating the sending time at a finer granularity e.g. microseconds, or nanoseconds.

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 evaluated the Split-Trust and Counter-Collusion configurations.

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).

- We used the testing application, made available by Drago et al. [18], to generate benchmarks, similar to those 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.

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

Table IV
CONFIGURATION COSTS

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, 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 the two configurations, indicated in tables II and III, 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 the configurations, are indicated in Table IV. 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.
- [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.

Files	File Size	End to End Latency (s)	CSP Sync latency (s)	CSP Sync latency component (%)
1	100KB	5.8	5.7	98.3
1	1MB	6.7	6.5	97.0
10	100KB	15.9	15.8	99.4
100	10KB	81.6	81.3	99.6

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	5.9	5.8	98.3
1	1MB	6.3	5.9	93.7
10	100KB	16.8	16.6	98.8
100	10KB	78.7	78.0	99.1

Table III

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

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

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

```

1) #!/bin/bash
2)
3) # The USER_DIR directory is created on the user's filesystem, and mounted onto the
4) # ENC_MODULE docker container (with read and write permissions). This is the
5) # directory through which the user updates (plaintext) files to be synced to the cloud,
6) # and receives updated (plaintext) files from the cloud.
7) # The ENC_CSP_INTERFACE directory is also created on the user's filesystem, and
8) # is mounted onto the ENC_MODULE and CSP_MODULE docker containers
9) # (with read and write permissions). This is the directory through which the two docker
10) # containers exchange encrypted files.
11) # These (USER_DIR and ENC_CSP_INTERFACE) are the only two directories, on
12) # the user's filesystem, that applications running within the docker containers are allowed to
13) # access.
14)
15)     BASE_DIR=~/framework_space/
16)     USER_DIR=${BASE_DIR}/plain_dir/
17)     ENC_CSP_INTERFACE=${BASE_DIR}/enc-csp_dir/
18)
19) # DECRYPTED_DIR is the name given to the USER_DIR directory, within the
20) # ENC_MODULE container. It is where the user's plaintext files are placed, before being
21) # encrypted and sent to the CSP, or after being received from the CSP, and decrypted.
22) # ENCRYPTED_DIR is the name given to the ENC_CSP_INTERFACE directory, within
23) # the ENC_MODULE container. It is where the user's ciphertext files are placed, after being
24) # encrypted.
25) # The TMP_ENC_DIR and TMP_DEC_DIR directories, are used to temporarily hold the files
26) # that are being encrypted/decrypted.
27) # The SCRIPTS_DIR directory is where the scripts that coordinate the encryption/decryption
28) # process are located.
29) # All of these directories are created in the ENC_MODULE container, and,
30) # (apart from the DECRYPTED_DIR and ENCRYPTED_DIR directories, which are shared)
31) # can only be accessed by applications running in this container.
32)
33)     DECRYPTED_DIR=/decrypted/
34)     ENCRYPTED_DIR=/encrypted/
35)     TMP_ENC_DIR=/enc_int/
36)     TMP_DEC_DIR=/dec_int/
37)     SCRIPTS_DIR=/scripts/
38)     ENC_MODULE="automated-enc_module"
39)     IMAGE="framework:enc_module"
40)
41) # CSP_DIR is the name given to the ENC_CSP_INTERFACE directory, within the
42) # CSP_MODULE container. This is the directory (shared with the ENC_MODULE container)
43) # which the CSP client monitors, and syncs with the CSP infrastructure.
44) # Note that the CSP client application running in the CSP_MODULE container has access only to
45) # the ENC_CSP_INTERFACE directory on the user's file system.
46)
47)     CSP_DIR=/csp/
48)     CSP_MODULE="automated-csp_module"
49)
50)     Enc_Image_Setup () {
51) # This line creates a docker container (with ubuntu as the operating system), running in the background (-d),
52) # with a pseudo terminal attached (-t), and the capability to receive input from the keyboard (-i)
53)     docker run -d -i -t --name="ENC_IMAGE" ubuntu
54)
55) # The following line runs a shell in the created docker container, and executes commands to:

```

```

56) # - create the temp directories used during the encryption/decryption process, as well as the directory for holding
57) # the scripts
58) # - update the list of Ubuntu repository packages ("apt-get update"), which is used when installing packages
59) # from the Ubuntu repositories via the apt-get install <package> command. The command "apt-get update"
60) # doesn't install anything in the container.
61) # Here, we are trusting that when this command is executed, a connection is established to the Ubuntu repositories,
62) # and the list of packages, with the correct signature for each package, is retrieved from the repository.
63) # - install the build-essential package from Ubuntu's repository ("apt-get -y install build-essential"), that is needed
64) # for building applications from source. We need this package to build the encryption application. This package
65) # consists of references to packages needed for building software on Ubuntu, including compilers (gcc and g++),
66) # GNU C library, and the MAKE utility.
67) # NOTE: If any of these packages is malicious, then it has access to the user's unencrypted files, and may be
68) # able to access the user's crypto key. Since we are assuming non-collusion for this configuration, this isn't an issue.
69) # The encryption container has no network access, and these packages are not installed in the CSP client container.
70) # Thus there's no way to leak the user's data/crypto key.
71) # - install the incron filesystem monitoring utility ("apt-get install incron")
72) # NOTE: This is a third party application, that has access to the user's unencrypted files, and also may be able to access
73) # the user's crypto key. However, since the encryption container has no network access, and this application isn't
74) # installed in the CSP client container, even if this application is malicious, it has no way to leak the user's data under
75) # this configuration.
76) # - enable the "root" user to use incron within the container ("echo 'root' > /etc/incron.allow")
77) # - create the filesystem monitoring rules for incron, and save them (in /var/spool/incron/root).
78) # The rules specify which directory to monitor (e.g. $DECRYPTED_DIR), which event to monitor for
79) # (e.g. IN_CLOSE_WRITE), and what action to take
80) # (e.g. $SCRIPTS_DIR/encrypt_script.pl $% '$@'$#' \ $ENCRYPTED_DIR $TMP_ENC_DIR
81) # $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' - this calls the "encrypt_script" (which we provide, and the user can audit),
82) # and passes it the name of the event {$%} that triggered incron, the url {@$#} of the file that triggered
83) # incron, the names of the directories used during the encryption/decryption process, as well as the url of the keyfile
84) # containing the encryption key/password).
85)
86)     docker exec ENC_IMAGE sh -c "mkdir $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR; \
87)     apt-get update; apt-get -y install build-essential; apt-get install incron; \
88)     echo 'root' > /etc/incron.allow; \
89)
90)     echo '$DECRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/encrypt_script.pl $% '$@'$#' \
91)     $ENCRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root; \
92)
93)     echo '$ENCRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/decrypt_script.pl $% '$@'$#' \
94)     $DECRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root"
95)
96) # The following three lines, stop the ENC_IMAGE container, save its image, and then delete the container.
97) # NOTE: We required network access to download and install the components to build the encryption application,
98) # however our security requirements are that the encryption container should have no network access. We therefore
99) # delete this container which was built with network access, and then later (in the Enc_Mod_Setup function) use the
100) # saved image (which has the libraries/tools require to build the encryption application) to create an encryption container
101) # that has no network access.
102)
103)     docker stop ENC_IMAGE
104)     docker commit ENC_IMAGE $IMAGE
105)     docker rm ENC_IMAGE
106) }
107)
108) Enc_Mod_Setup () {
109) # This line creates the encryption container, from the previously saved image, with the two interface directories,
110) # USER_DIR and ENC-CSP_INTERFACE mounted in the container (-v), and with no network access (-net=none).
111)
112)     docker run -d -i -t --net=none -v $1:$DECRYPTED_DIR -v $2:$ENCRYPTED_DIR --name=$3 $IMAGE
113)

```

```

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 that triggered
14 # 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 of the
18 # the directories used to temporarily hold files during the encryption/
19 # decryption process.
20 # - The "passFile" variable is assigned the full path of the file containing
21 # 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 #
34 # It receives the url of the file to be encrypted, its name,
35 # and the encryption password/key.
36 # We then specify the name of the output file to be created
37 # after encryption(path + extension), call the encryption
38 # application, and return the full path to the encrypted file.
39 #
40 # Note that the encrypted file is placed in the tmpEnc
41 # directory.
42 # This helps the "decrypt_script" distinguish between the case
43 # where a file entering the "Enc" directory, is from the CSP and
44 # needs to be decrypted, or is from the user and has just been encrypted.
45 # We need to distinguish these two cases in order to avoid
46 # getting into an infinite loop of encryption/decryption over the same file.
47 #
48 # After this function returns, the encrypted file is copied
49 # to the encrypted directory.
50
51     my $fileURL = $_[0];
52     my $fileName = $_[1];
53     my $pWord = $_[2];
54     my $ext = '.aes';
55     my $output = "$tmpEnc"."$fileName"."$ext";
56     `aescrypt -e -p $pWord -o $output $fileURL`;
```

```

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

```

```
115      # the encryption password/key to the encryption function.
116      # The encryption function encrypts the file , then returns the full path
117      # to the encrypted file .
118      # The encrypted file is then copied to the encrypted directory , and this
119      # script's work is done.
120
121      my $outFile = Encrypt($url , $fname , $passWord);
122      copy($outFile , $encrypted);
123      }
124  }
125
126  Main() ;
```


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

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

```

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

```

```
115      # The decryption function decrypts the file , then returns the full path
116      # to the decrypted file .
117      # The decrypted file is then copied to the decrypted directory , and this
118      # script's work is done .
119
120      my $outFile = Decrypt($url , $fname , $passWord);
121      copy($outFile , $decrypted);
122      }
123  }
124
125  Main() ;
```

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

```
1) #!/bin/bash
2)
3) # The USER_DIR directory is created on the user's filesystem, and mounted onto the ENC_MODULE docker
4) # container (with read and write permissions). This is the directory through which the user updates (plaintext) files
5) # to be synced to the cloud, and receives updated (plaintext) files from the cloud.
6) # The ENC1_ENC2_INTERFACE directory is also created on the user's filesystem, and is mounted onto the
7) # ENC_MODULE and ENC_MODULE2 docker containers (with read and write permissions). This is the
8) # directory through which these two docker containers exchange encrypted files.
9) # The ENC2_CSP_INTERFACE directory is also created on the user's filesystem, and is mounted onto the
10) # ENC_MODULE and CSP_MODULE docker containers (with read and write permissions). This is the
11) # directory through which these two docker containers # exchange encrypted files. The USER_DIR,
12) # ENC1_ENC2_INTERFACE, and ENC2_CSP_INTERFACE directories, are the only directories, on
13) # the user's filesystem, that applications running within the docker containers are allowed to access.
14)
15)     BASE_DIR=~/.framework_space/
16)     USER_DIR=${BASE_DIR}/plain_dir/
17)     ENC1_ENC2_INTERFACE=${BASE_DIR}/enc1-enc2_dir/
18)     ENC2_CSP_INTERFACE=${BASE_DIR}/enc2-csp_dir/
19)
20) # DECRYPTED_DIR is the name given to the USER_DIR directory, within the ENC_MODULE container.
21) # It is where the user's plaintext files are placed, before being encrypted and sent to ENC_MODULE2,
22) # or after being received from ENC_MODULE2, and decrypted.
23) # ENCRYPTED_DIR is the name given to the ENC1_ENC2_INTERFACE directory, within the
24) # ENC_MODULE container. It is where the user's ciphertext files are placed, after being encrypted
25) # by the encryption application in ENC_MODULE. In the ENC_MODULE2 container, the
26) # ENC1_ENC2_INTERFACE directory is mounted as DECRYPTED_DIR, and the
27) # ENC2_CSP_INTERFACE directory is mounted as ENCRYPTED_DIR. In both containers,
28) # ENC_MODULE and ENC_MODULE2, the TMP_ENC_DIR and TMP_DEC_DIR directories, are used to
29) # temporarily hold the files that are being encrypted/decrypted. The SCRIPTS_DIR directory is where the scripts
30) # that coordinate the encryption/decryption process are located.
31) # NOTE: The applications running in each of these containers can only access the directories(and files)in their
32) # respective containers. Apart from being able to access (read and write) the three directories, USER_DIR,
33) # ENC1_ENC2_INTERFACE, and ENC2_CSP_INTERFACE, that are mounted from the user's filesystem,
34) # the applications running in each of these containers have no access to any other part of the user's filesystem.
35)
36)     DECRYPTED_DIR=/decrypted/
37)     ENCRYPTED_DIR=/encrypted/
38)     TMP_ENC_DIR=/enc_int/
39)     TMP_DEC_DIR=/dec_int/
40)     SCRIPTS_DIR=/scripts/
41)     ENC_MODULE="automated-enc_module"
42)     ENC_MODULE2="automated-enc_module2"
43)     IMAGE="framework:enc_module"
44)
45) # CSP_DIR is the name given to the ENC2_CSP_INTERFACE directory, within the CSP_MODULE container.
46) # This is the directory (shared with the ENC_MODULE2 container) which the CSP client monitors, and syncs with the
47) # CSP infrastructure.
48) # Note that the CSP client application running in the CSP_MODULE container has access only to the
49) # ENC2_CSP_INTERFACE directory on the user's file system.
50)
51)     CSP_DIR=/csp/
52)     CSP_MODULE="automated-csp_module"
53)
54)     Enc_Image_Setup () {
55)
56) # This line creates a docker container (with ubuntu as the operating system), running in the background (-d),
```

```

57) # with a pseudo terminal attached (-t), and the capability to receive input from the keyboard (-i)
58)
59)     docker run -d -i -t --name="ENC_IMAGE" ubuntu
60)
61) # The following line runs a shell in the created docker container, and executes commands to:
62) # - create the temp directories used during the encryption/decryption process, as well as the directory for holding
63) # the scripts
64) # - update the list of Ubuntu repository packages ("apt-get update"), which is used when installing packages from
65) # the Ubuntu repositories via the apt-get install <package> command. The command "apt-get update" doesn't install
66) # anything in the container. Here, we are trusting that when this command is executed, a connection is established to
67) # the Ubuntu repositories, and the list of packages, with the correct signature for each package, is retrieved from
68) # the repository
69) # - install the build-essential package from Ubuntu's repository ("apt-get -y install build-essential"), that is needed for
70) # building applications from source. We need this package to build the encryption application. This package consists of
71) # references to packages needed for building software on Ubuntu, including compilers (gcc and g++), GNU C library,
72) # and the MAKE utility
73) # NOTE: We assume that these packages are not malicious, since they are obtained from the Ubuntu repository, and we
74) # are trusting our operating system, and by extension the OS provider. However, if any of these packages is malicious,
75) # and it colludes with the CSP client (under this configuration, we assume some malicious components may collude),
76) # then
77) # this would represent a serious vulnerability. Since, these packages are installed in each of the encryption containers,
78) # and thus would be able to access the user's unencrypted data, as well as cryptographic keys.
79) # - install the incron filesystem monitoring utility ("apt-get install incron")
80) # NOTE: This is a third party application, that has access to the user's unencrypted files, and also may be able to access
81) # the user's crypto keys. Again, under this configuration, we assume some malicious components may collude, therefore
82) # if this filesystem monitoring tool is malicious, and colludes with the CSP client application, it would be able to leak
83) # the user's data, or cryptographic keys. We assume that this application is trusted not to collude with the CSP client
84) # application.
85) # - enable the "root" user to use incron within the container ("echo 'root' > /etc/incron.allow")
86) # - create the filesystem monitoring rules for incron, and save them (in /var/spool/incron/root).
87) # The rules specify which directory to monitor (e.g. $DECRYPTED_DIR), which event to monitor for
88) # (e.g. IN_CLOSE_WRITE), and what action to take
89) # (e.g. $SCRIPTS_DIR/encrypt_script.pl $% '$@'$#' $ENCRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR
90) # $SCRIPTS_DIR/keyfile' – this calls the "encrypt_script" (which we provide, and the user can audit), and passes it
91) # the name of the event {$%} that triggered incron, the url {@$#} of the file that triggered incron, the names of the
92) # directories used during the encryption/decryption process, as well as the url of the keyfile containing the encryption
93) # key/password).
94)
95)     docker exec ENC_IMAGE sh -c "mkdir $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR; \
96) apt-get update; apt-get -y install build-essential; apt-get install incron; \
97) echo 'root' > /etc/incron.allow; \
98)
99)     echo '$DECRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/encrypt_script.pl $% '$@'$#' \
100) $ENCRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root; \
101)
102)     echo '$ENCRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/decrypt_script.pl $% '$@'$#' \
103) $DECRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root"
104)
105) # The following three lines, stop the ENC_IMAGE container, save its image, and then delete the container.
106) # NOTE: We required network access to download and install the components to build the encryption application,
107) # however our security requirements are that the encryption container should have no network access. We therefore
108) # delete this container which was built with network access, and then later (in the Enc_Mod_Setup function) use the
109) # saved image (which has the libraries/tools require to build the encryption application) to create an encryption container
110) # that has no network access.
111)
112)     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)

```
1) #!/bin/bash
2)
3) # NOTE: The applications running in each of the five containers, INIT_MODULE, ENC_MODULE,
4) # ENC_MODULE2, VERIFICATION_MODULE, and the CSP_MODULE, can only access the
5) # directories(and files)in their respective containers. Apart from being able to access
6) # (read and write) the directories, USER_DIR, INIT_ENC1_INTERFACE,
7) # INIT_ENC2_INTERFACE, ENC1_VER_INTERFACE, ENC2_VER_INTERFACE, and
8) # VER_CSP_INTERFACE, that are mounted from the user's filesystem, the applications running in each
9) # of these containers have no access to any other part of the user's filesystem.
10)
11) BASE_DIR=~/.framework_space/
12) USER_DIR=${BASE_DIR}/plain_dir/
13) INIT_ENC1_INTERFACE=${BASE_DIR}/init-enc1_dir/
14) INIT_ENC2_INTERFACE=${BASE_DIR}/init-enc2_dir/
15) VER_CSP_INTERFACE=${BASE_DIR}/ver-csp_dir/
16) ENC_MODULE="automated-enc_module"
17) ENC_MODULE2="automated-enc_module2"
18) CSP_MODULE="automated-csp_module"
19) VERIFICATION_MODULE="automated-ver_module"
20) INIT_MODULE="automated-init_module"
21) ENC_IMAGE="framework:enc_module"
22) AUX_IMAGE="framework:aux_module"
23) DECRYPTED_DIR=/decrypted/
24) ENCRYPTED_DIR=/encrypted/
25) ENCRYPTED_DIR2=/encrypted2/
26) TMP_ENC_DIR=/enc_int/
27) TMP_DEC_DIR=/dec_int/
28) SCRIPTS_DIR=/scripts/
29) CSP_DIR=/csp/
30)
31) Enc_Image_Setup () {
32)
33) # This line creates a docker container (with ubuntu as the operating system), running in the background (-d),
34) # with a pseudo terminal attached (-t), and the capability to receive input from the keyboard (-i)
35)
36) docker run -d -i -t --name="ENC_IMAGE" ubuntu
37)
38) # The following line runs a shell in the created docker container, and executes commands to:
39) # - create the temp directories used during the encryption/decryption process, as well as the directory for holding
40) # the scripts
41) # - update the list of Ubuntu repository packages ("apt-get update"), which is used when installing packages from
42) # the Ubuntu repositories via the apt-get install <package> command. The command "apt-get update" doesn't install
43) # anything in the container. Here, we are trusting that when this command is executed, a connection is established to
44) # the Ubuntu repositories, and the list of packages, with the correct signature for each package, is retrieved from
45) # the repository.
46) # - install the build-essential package from Ubuntu's repository # ("apt-get -y install build-essential"), that is needed for
47) # building applications from source. We need this package to build the encryption application. This package consists
48) # of references to packages needed for building software on Ubuntu, including compilers (gcc and g++), GNU C library,
49) # and the MAKE utility.
50) # NOTE: We assume that these packages are not malicious, since they are obtained from the Ubuntu repository, and
51) # we are trusting our operating system, and by extension the OS provider. However, even if any of these packages is
52) # malicious, and it colludes with the CSP client (under this configuration, we assume some malicious components may
53) # collude), it would only be able to leak the user's data, or cryptographic keys via some covert channel that isn't
54) # addressed by our design. Because, though these packages are installed in each of the encryption containers, and thus
55) # would be able to access the user's unencrypted data, as well as cryptographic keys, they are not installed in the
56) # Verification container.
```

```

57) # - install the incron filesystem monitoring utility ("apt-get install incron")
58) # NOTE: This is a third party application, that has access to the user's unencrypted files, and also may be able to access
59) # the user's crypto keys. Under this configuration, we assume some malicious components may collude, therefore if this
60) # filesystem monitoring tool is malicious, and colludes with the CSP client application, it would be able to leak the
61) # user's data, or cryptographic keys. We assume that this application is trusted not to collude with the CSP client
62) # application.
63) # - enable the "root" user to use incron within the container ("echo 'root' > /etc/incron.allow")
64) # - create the filesystem monitoring rules for incron, and save them (in /var/spool/incron/root)
65) # The rules specify which directory to monitor (e.g. $DECRYPTED_DIR), which event to monitor for
66) # (e.g. IN_CLOSE_WRITE), and what action to take
67) # (e.g. $SCRIPTS_DIR/encrypt_script.pl % $'@$'#$' $ENCRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR
68) # $SCRIPTS_DIR/keyfile' – this calls the "encrypt_script" (which we provide, and the user can audit), and passes it the
69) # name of the event { $% } that triggered incron, the url { @$# } of the file that triggered incron, the names of the
70) # directories used during the encryption/decryption process, as well as the url of the keyfile containing the encryption
71) # key/password).
72) # For this configuration, we require the two encryption applications to use the same key and initialization
73) # vector when encrypting the user's files, so that we can verify that the resulting ciphertext is the same. We therefore
74) # generate the key and IV during this stage, and store the values in the file "keyfile". This is achieved
75) # with the following commands:
76) # dd if=/dev/urandom of=$SCRIPTS_DIR/keyfile.data bs=1024 count=100;
77) # md5sum $SCRIPTS_DIR/keyfile.data | cut -d ' ' -f1 > $SCRIPTS_DIR/keyfile
78) #
79) # 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
80) # the scripts (encrypt_script.pl and decrypt_script.pl) in the encryption containers, and then passed to the encryption
81) # application during encryption/decryption. A new value is generated randomly each time this configuration script is
82) # run, and the user can also edit the keyfile, to manually change/update this value. Our goal here is to generate a default
83) # key and IV, that are hard to guess and that can be used by the user for encryption/decryption of files, as well as give
84) # the user the flexibility to change the key/IV values whenever he/she wants to.
85) #
86) # So when the Enc_Mod_Setup() function is called to create the encryption containers from this image, they will
87) # both have the same keyfile copy in the SCRIPTS_DIR. In order to share files with multiple devices, each running this
88) # configuration, the user will need to securely transfer one of the "keyfile" copies from an encryption container, to the
89) # encryption containers created on the other devices.
90)
91)     docker exec ENC_IMAGE sh -c "mkdir $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR; \
92)     apt-get update; apt-get -y install build-essential; apt-get install incron; \
93)     echo 'root' > /etc/incron.allow; \
94)
95)     echo '$DECRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/encrypt_script.pl % $'@$'#$' \
96)     $ENCRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root; \
97)
98)     echo '$ENCRYPTED_DIR IN_CLOSE_WRITE $SCRIPTS_DIR/decrypt_script.pl % $'@$'#$' \
99)     $DECRYPTED_DIR $TMP_ENC_DIR $TMP_DEC_DIR $SCRIPTS_DIR/keyfile' >> /var/spool/incron/root; \
100)
101)     dd if=/dev/urandom of=$SCRIPTS_DIR/keyfile.data bs=1024 count=100; \
102)     md5sum $SCRIPTS_DIR/keyfile.data | cut -d ' ' -f1 > $SCRIPTS_DIR/keyfile"
103)
104) # The following three lines, stop the ENC_IMAGE container, save its image, and then delete the container.
105) # NOTE: We required network access to download and install the components to build the encryption application,
106) # however our security requirements are that the encryption container should have no network access. We therefore
107) # delete this container which was built with network access, and then later (in the Enc_Mod_Setup function) use the
108) # saved image (which has the libraries/tools require to build the encryption application) to create an encryption container
109) # that has no network access.
110)
111)     docker stop ENC_IMAGE
112)     docker commit ENC_IMAGE $1
113)     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 VERIFICATION 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 VERIFICATION 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 triggered
13 # 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 paths of the
17 # the directories used to temporarily hold files during the encryption/
18 # decryption process.
19 # – The "keyFile" variable is assigned the full path of the file containing
20 # 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.
34 #
35 # It receives the url of the file to be encrypted, its name,
36 # and the encryption key and IV.
37 # We then specify the name(path + extension) of the output file
38 # to be created after encryption, call the encryption
39 # application, and return the full path to the encrypted file.
40 # Note that the encrypted file is placed in the tmpEnc
41 # directory. It is later (after this function returns) copied
42 # to the encrypted directory.
43
44     my $fileURL = $_[0];
45     my $fileName = $_[1];
46     my $encKey = $_[2];
47     my $encIV = $_[3];
48     my $ext = '.enc';
49     my $file = "$tmpEnc"."$fileName";
50     my $output = "$tmpEnc"."$fileName"."$ext";
51     copy($fileURL, $tmpEnc);
52     'sisicrypt -e -f $file -k $encKey -i $encIV';
53     unlink $file;
54     return $output;
55 }
56
```

```

57 sub Main {
58
59 # This script monitors the "decrypted" directory via which, plaintext
60 # files are exchanged with the INIT container.
61 # When a file, to be encrypted (received from the INIT container), or
62 # which has just been decrypted (to be sent to the INIT container),
63 # lands on this directory, this script is triggered, and the url of the
64 # file is passed to this script.
65
66         chomp($url);
67
68 # we extract the file name from the url
69
70         my $fname = fileparse($url);
71         my $compareFile = $tmpDec . $fname;
72
73 # We check if the file that triggered the script has a similar copy
74 # in the "tmpDec" directory. If a similar copy exists in the tmpDec
75 # directory, it means that the file should not be encrypted, since
76 # it has just been decrypted.
77 # When the "decrypt_script", in charge of decryption, decrypts a file,
78 # it places a copy of the decrypted plaintext file in the tmpDec
79 # directory, before copying the file to the "decrypted" directory.
80 # This helps distinguish between the case where a file entering the "decrypted"
81 # directory(which triggers this script), is from the CSP and has just
82 # been decrypted, or is from the user and needs to be encrypted before being
83 # sent to the CSP.
84
85         my $fileCheck = compare($url, $compareFile);
86         if ($fileCheck == 0) {
87
88             # if a similar copy of the file exists in the tmpDec directory,
89             # we delete the copy of the file in the tmpDec directory, and exit.
90             # There's nothing else to do.
91
92             unlink $compareFile;
93         } else {
94
95             # otherwise, we open the file containing the encryption key and IV,
96             # extract the two values, and close the file.
97
98             open(KEYFILE, "<$keyFile") || die("failed to open $keyFile\n");
99             if ((read KEYFILE, $key, 16) != 16) {die("failed to read key\n")};
100            if ((read KEYFILE, $iv, 16) != 16) {die("failed to read iv\n")};
101            close(KEYFILE);
102
103            # we then pass the url of the file to be encrypted, the filename, and
104            # the encryption key and IV, to the encryption function.
105            # The encryption function encrypts the file, then returns the full path
106            # to the encrypted file.
107            # The encrypted file is then copied to the encrypted directory, and this
108            # script's work is done.
109
110            my $outFile = Encrypt($url, $fname, $key, $iv);
111            copy($outFile, $encrypted);
112        }
113    }
114

```

```
115 Main() ;
```


G. Counter-CovertChannels Configuration Decryption Script (decrypt_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 triggered
13 # this script.
14 # - The variable "decrypted" is assigned the full path of the directory
15 # where decrypted files are stored
16 # - The "tmpEnc" and "tmpDec" directories are assigned the full paths of the
17 # the directories used to temporarily hold files during the encryption/
18 # decryption process.
19 # - The "keyFile" variable is assigned the full path of the file containing
20 # the key and initialisation vector (IV) used for decryption
21
22 my $url = $ARGV[1];
23 my $decrypted = $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 Decrypt{
31
32 # This function should be modified, to suit the requirements
33 # of the encryption application being used.
34 #
35 # It receives the url of the file to be decrypted, its name,
36 # and the decryption key and IV.
37 # We then specify the name(path) of the output file to be created
38 # after decryption, call the encryption
39 # application, and return the full path to the decrypted file.
40 # Note that the decrypted file is placed in the tmpDec
41 # directory. It is later (after this function returns) copied
42 # to the "decrypted" directory.
43
44     my $fileURL = $_[0];
45     my $fileName = $_[1];
46     my $encKey = $_[2];
47     my $encIV = $_[3];
48     my $file = "$tmpDec"."$fileName";
49     my $output = "$tmpDec"."$fileName";
50     $output =~ s/.enc//;
51     copy($fileURL, $tmpDec);
52     'sisicrypt -d -f $file -k $encKey -i $encIV';
53     unlink $file;
54     return $output;
55 }
56
```

```

57 sub Main {
58
59 # This script monitors the "encrypted" directory which holds
60 # ciphertext files.
61 # When an encrypted file , received from, or being sent to, the
62 # VERFICATION container , lands in this directory , this script
63 # is triggered , and the url of the file is passed to this script.
64
65         chomp( $url );
66
67 # we extract the file name from the url
68
69         my $fname = fileparse( $url );
70         my $compareFile = $tmpEnc . $fname;
71
72 # We check if the file that triggered the script has a similar copy
73 # in the tmpEnc directory. If a similar copy exists in the tmpEnc
74 # directory , it means that the file should not be decrypted , since
75 # it has just been encrypted.
76 # When the "encrypt_script", in charge of encryption , encrypts a file ,
77 # it places a copy of the encrypted file in the tmpEnc
78 # directory , before copying the file to the "encrypted" directory.
79 # This helps distinguish between the case where a file entering the
80 # "encrypted" directory(which triggers this script), is from the CSP and
81 # needs to be decrypted , or is from the user and has just been encrypted.
82
83         my $fileCheck = compare( $url , $compareFile );
84         if ( $fileCheck == 0 ) {
85
86             # if a similar copy of the file exists in the tmpEnc directory ,
87             # we delete the copy of the file in the tmpEnc directory , and exit.
88             # There's nothing else to do.
89
90             unlink $compareFile;
91         } else {
92
93             # otherwise , we open the file containing the decryption key and IV,
94             # extract the two values , and close the file.
95
96             open(KEYFILE, "<$keyFile") || die("failed to open $keyFile\n");
97             if ((read KEYFILE, $key, 16) != 16) {die("failed to read key\n")};
98             if ((read KEYFILE, $iv, 16) != 16) {die("failed to read iv\n")};
99             close(KEYFILE);
100
101             # we then pass the url of the file to be decrypted , the filename , and
102             # the decryption key and IV, to the Decrypt function.
103             # The Decrypt function decrypts the file , then returns the full path
104             # to the decrypted file .
105             # The decrypted file is then copied to the "decrypted" directory , and this
106             # script's work is done.
107
108             my $outFile = Decrypt( $url , $fname , $key , $iv );
109             copy( $outFile , $decrypted );
110         }
111     }
112
113     Main( );

```

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 crypto 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
84                 outFile!");
85         }
86         memset(buffer, 0, BUFFERLENGTH);
87         if (ferror(inFp)){
88             Error("I/O error: Didn't complete reading from input file!\n"
89                 );
90         }
91
92         /* function to destroy encryption context, and shutdown crypto library */
93
94         fclose(inFp);
95         fclose(outFp);
96         free(outFile);
97     }
98
99     void DecryptFile(char *fileName, char *key, char *iv) {
100         int status;
101         char    buffer[BUFFERLENGTH];
102         char *outFile;
103         FILE *inFp = NULL, *outFp = NULL;
104
105         outFile = calloc((strlen(fileName) + 1 - 4), 1);
106         if (outFile == NULL) {
107             Error("Failed to allocate memory for outFile!\n");
108         }
109         strncpy(outFile, fileName, (strlen(fileName) - 4));
110
111         outFp = fopen(outFile, "w");
112         if (outFp == NULL) {
113             Error("failed to open output file for writing.");
114         }

```

```

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
128                 outFile!");
129         }
130         memset(buffer, 0, BUFFERLENGTH);
131     }
132     if (ferror(inFp)){
133         Error("I/O error: Didn't complete reading from input file!\n");
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
144         key> -i <initialisation vector>\n");
145     printf("To Decrypt file:\t sisicrypt -d -f <filename> -k <encryption key> -i
146         <initialisation vector>\n");
147     exit(EXIT_FAILURE);
148 }
149
150 void Error(char * errorMessage){
151     fprintf(stderr, "%s\n", errorMessage);
152     exit(EXIT_FAILURE);
153 }

```

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 triggered
13 # this script.
14 # – The variable "encrypted" is assigned the full path of the directory
15 # that interfaces with the ENC_MODULE container
16 # – The variable "encrypted2" is assigned the full path of the directory
17 # that interfaces with the ENC_MODULE2 container
18
19 my $url = $ARGV[1];
20 my $encrypted = $ARGV[2];
21 my $encrypted2 = $ARGV[3];
22
23 sub Main {
24
25 # This script monitors the "decrypted" directory in the INIT
26 # container, which interfaces with the User's directory.
27 # When a file, to be synced to the CSP (created, or updated by the
28 # user), or which has just been received from the CSP,
29 # lands on this directory, this script is triggered, by the matching rule
30 # specified in the incrontab file (/var/spool/incron/root),
31 # and this matching rule passes the url of the file to this script.
32 #
33 # Note that the rules, specifying the directories to be monitored by incron,
34 # the events to monitor, and the actions to take when the events occur,
35 # are created during the configuration setup phase
36 # (counter-covertchannels_config.sh — specifically in the Aux_Image_Setup ()
37 # function)
38
39         chomp($url);
40
41 # we extract the file name from the url
42
43         my $fname = fileparse($url);
44         my $compareFile = $encrypted . $fname;
45         my $compareFile2 = $encrypted2 . $fname;
46
47 # We check if the file that triggered the script has a similar copy
48 # in the "encrypted" and "encrypted2" directories. If a similar copy
49 # exists, it means that the file should not be sent to the two encryption
50 # containers(ENC_MODULE and ENC_MODULE2), since it has just been received from
51 # the "encrypted" directory, and is present in both "encrypted" and
52 # "encrypted2" directories.
53 # This helps distinguish between the case where a file entering the
54 # "decrypted" directory(which triggers this script), is from the CSP
55 # and has just been decrypted, or is from the user and needs to be
56 # encrypted before being sent to the CSP. We need to distinguish
```



```

57 # these two cases to avoid getting into an infinite loop.
58
59 my $fileCheck = compare($url, $compareFile);
60 my $fileCheck2 = compare($url, $compareFile2);
61 if (($fileCheck != 0) && ($fileCheck2 != 0)) {
62
63     # If a similar copy isn't found in the "encrypted" directory,
64     # it means that the file is to be encrypted, then sent to the CSP.
65     # We thus copy the file to the directories that interface with
66     # the two encryption containers(ENC_MODULE and ENC_MODULE2).
67     # The objective is to have the same file encrypted by the two
68     # encryption applications in ENC_MODULE and ENC_MODULE2 respectively,
69     # using the same key and initialization vector, so that we can verify
70     # whether they are both producing the same output
71
72     copy($url, $encrypted);
73     copy($url, $encrypted2);
74 }
75 }
76
77 Main();

```

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 that triggered
13 # this script. # – The variable "encrypted2" is assigned the full path of the
   directory
14 # that interfaces with the ENC_MODULE2 container
15 # – The variable "plainDir" is assigned the full path of the directory
16 # that interfaces with the user's directory (USER_DIR)
17
18 my $url = $ARGV[1];
19 my $encrypted2 = $ARGV[2];
20 my $plainDir = $ARGV[3];
21
22 sub Main {
23
24 # This script monitors the "encrypted" directory in the INIT
25 # container, which interfaces with the encryption container
26 # ENC_MODULE.
27 # When a file, to be synced to the CSP (created, or updated by the
28 # user), or which has just been received from the CSP,
29 # lands on this directory, this script is triggered, by the matching rule
30 # specified in the incrontab file (/var/spool/incron/root),
31 # and this matching rule passes the url of the file to this script.
32 #
33 # Note that the rules, specifying the directories to be monitored by incron,
34 # the events to monitor, and the actions to take when the events occur,
35 # are created during the configuration setup phase
36 # (counter-covertchannels_config.sh — specifically in the Aux_Image_Setup ()
37 # function)
38
39         chomp($url);
40
41 # we extract the file name from the url
42
43         my $fname = fileparse($url);
44         my $compareFile = $plainDir . $fname;
45
46 # We check if the file that triggered the script has a similar copy
47 # in the plainDir directory. If a similar copy exists, it means
48 # that the file should not be sent to the user's directory, since it
49 # has just been received from the user, and is being synced to the CSP.
50 # This helps distinguish between the case where a file entering the
51 # "encrypted" directory(which triggers this script), is from the CSP
52 # and has just been decrypted, or is from the user and needs to be
53 # encrypted before being sent to the CSP. We need to distinguish these
54 # two cases in order to avoid getting into an infinite loop.
55
```

```

56      my $fileCheck = compare($url , $compareFile);
57      if($fileCheck != 0) {
58
59          # If a similar copy isn't found in the plainDir directory ,
60          # it means that the file has been received from the CSP, and is to
61          # be copied to the user's directory.
62
63          copy($url , $plainDir);
64      }
65  }
66
67  Main() ;

```

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 that triggered
13 # this script.
14 # – The variable "encrypted" is assigned the full path of the directory
15 # that interfaces with the ENC_MODULE container
16 # – The variable "plainDir" is assigned the full path of the directory
17 # that interfaces with the user's directory(USER_DIR)
18
19 my $url = $ARGV[1];
20 my $encrypted = $ARGV[2];
21 my $plainDir = $ARGV[3];
22
23 sub Main {
24
25 # This script monitors the "encrypted2" directory in the INIT
26 # container, which interfaces with the encryption container
27 # ENC_MODULE2.
28 # When a file, to be synced to the CSP (created, or updated by the
29 # user), or which has just been received from the CSP,
30 # lands on this directory, this script is triggered, by the matching rule
31 # specified in the incrontab file (/var/spool/incron/root),
32 # and this matching rule passes the url of the file to this script.
33 #
34 # Note that the rules, specifying the directories to be monitored by incron,
35 # the events to monitor, and the actions to take when the events occur,
36 # are created during the configuration setup phase
37 # (counter-covertchannels_config.sh — specifically in the Aux_Image_Setup ()
38 # function)
39
40         chomp($url);
41
42 # we extract the file name from the url
43
44         my $fname = fileparse($url);
45         my $compareFile = $plainDir . $fname;
46
47 # We check if the file that triggered the script has a similar copy
48 # in the plainDir directory. If a similar copy exists, it means
49 # that the file should not be sent to the user's directory, since it
50 # has just been received from the user, and is being synced to the CSP.
51 # This helps distinguish between the case where a file entering the
52 # "encrypted2" directory(which triggers this script), is from the CSP
53 # and has just been decrypted, or is from the user and needs to be
54 # encrypted before being sent to the CSP.
55 # We need to distinguish these two cases in order to avoid getting into
56 # an infinite loop.
```

```

57
58     my $fileCheck = compare($url , $compareFile);
59     if($fileCheck != 0) {
60
61         # If a similar copy isn't found in the plainDir directory ,
62         # it means that the file has been received from the CSP.
63         # We therefore copy it to the user's directory.
64
65         copy($url , $plainDir);
66     }
67 }
68
69 Main();

```

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 triggered
13 # this script. # – The variable "encrypted" is assigned the full path of the
   directory
14 # that interfaces with the ENC_MODULE container
15 # – The variable "encrypted2" is assigned the full path of the directory
16 # that interfaces with the ENC_MODULE2 container
17
18 my $url = $ARGV[1];
19 my $encrypted = $ARGV[2];
20 my $encrypted2 = $ARGV[3];
21
22 sub Main {
23
24 # This script monitors the "decrypted" directory in the VERIFICATION
25 # container, which interfaces with the CSP's directory.
26 # When a file, to be synced to the CSP (created, or updated by the
27 # user), or which has just been received from the CSP,
28 # lands on this directory, this script is triggered, by the matching rule
29 # specified in the incrontab file (/var/spool/incron/root),
30 # and this matching rule passes the url of the file, as well as the
31 # directory arguments, to this script.
32 #
33 # Note that the rules, specifying the directories to be monitored by incron,
34 # the events to monitor, and the actions to take when the events occur,
35 # are created during the configuration setup phase
36 # (counter-covertchannels_config.sh — specifically in the Aux_Image_Setup ()
37 # function)
38
39         chomp($url);
40
41 # we extract the file name from the url
42
43         my $fname = fileparse($url);
44         my $compareFile = $encrypted . $fname;
45         my $compareFile2 = $encrypted2 . $fname;
46
47 # We check if the file that triggered the script has a similar copy
48 # in the "encrypted" or "encrypted2" directories, which interface with
49 # the ENC_MODULE and ENC_MODULE2 containers.
50 # If a similar copy exists, it means that the file should
51 # not be sent to the two encryption containers(ENC_MODULE and ENC_MODULE2),
52 # since it has just been received from the "encrypted" directory, and is
53 # present in both "encrypted" and "encrypted2" directories.
54 #
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 needs to be decrypted before being sent to the user, or is from
58 # the user. We need to distinguish these two cases to avoid getting into
59 # an infinite loop.
60
61     my $fileCheck = compare($url, $compareFile);
62     my $fileCheck2 = compare($url, $compareFile2);
63     if (($fileCheck != 0) && ($fileCheck2 != 0)) {
64
65         # If a similar copy isn't found in the "encrypted" directory,
66         # it means that the file is to be decrypted, then sent to the user.
67         # We thus copy the file to the directories that interface with
68         # the two encryption containers(ENC_MODULE and ENC_MODULE2)
69
70         copy($url, $encrypted);
71         copy($url, $encrypted2);
72     }
73 }
74
75 Main();

```

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 that triggered
13 # this script.
14 # - The variable "encrypted" is assigned the full path of the directory
15 # 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 file we use
19 # 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 my $dispatchTime;
26
27 sub FileVerification {
28
29 # This function takes in 3 input parameters, the URL of the file
30 # that triggered this script, the name of the file, and the path
31 # to the "encrypted2" directory.
32 # It then checks whether a similar (content) copy of the file that
33 # triggered this script, exists in the "encrypted2" directory.
34 # If it finds one, it copies the file that triggered this script to
35 # the "csp" directory.
36 # It returns a status code indicating whether or not a similar file
37 # was found.
38
39     my $fileURL = $_[0];
40     my $fileName = $_[1];
41     my $dir = $_[2];
42     my $compareFile = $dir . $fileName;
43     my $compareStatus = compare($fileURL, $compareFile);
44
45     if ($compareStatus == 0) {
46         my $currentTime = 'date +%M:%S' ;
47         chomp($currentTime);
48         my @time = split(':', $currentTime);
49         my $minute = $time[0];
50         my $seconds = $time[1];
51
52         if (($minute % 2) == 0) {
53             $dispatchTime = 120;
54         } else {
55             $dispatchTime = 180;
56         }
57     }
```



```

57
58         # Assigning the dispatchTime value as above ensures that the file
59         # is always sent out at the top of an even minute. This prevents a
60         # malicious encryption application from leaking information to the
61         # CSP client by modulating the file send time between odd and even
62         # minutes.
63
64         my $dispatchCountDown = $dispatchTime - $seconds;
65         sleep ($dispatchCountDown);
66         copy($fileURL, $csp);
67     }
68     return $compareStatus;
69 }
70
71 sub Main {
72
73     # This script monitors the "encrypted" directory in the VERIFICATION
74     # container, which interfaces with the encryption container ENC_MODULE.
75     # When a file, to be synced to the CSP, or which has just been received
76     # from the CSP, lands on this directory, this script is triggered, by the
77     # matching rule specified in the incrontab file (/var/spool/incron/root).
78     # The matching rule passes the url of the file to this script.
79     #
80     # Note that the rules, specifying the directories to be monitored by incron,
81     # the events to monitor, and the actions to take when the events occur,
82     # are created during the configuration setup phase
83     # (counter-covertchannels_config.sh — specifically in the Aux_Image_Setup ()
84     # function)
85
86     chomp ($url);
87
88     # we extract the file name from the url
89
90     my $fname = fileparse($url);
91     my $compareFile = $csp . $fname;
92
93     # We check if the file that triggered the script has a similar copy
94     # in the "csp" directory. If a similar copy exists, it means
95     # that the file should not be sent to the CSP's directory, since it
96     # has just been received from the CSP, and is being synced to the user.
97     # This helps distinguish between the case where a file entering the
98     # "encrypted" directory(which triggers this script), is from the CSP
99     # and needs to be decrypted before being sent to the user, or is from
100    # the user. We need to distinguish these two cases in order to avoid
101    # getting into an infinite loop.
102
103    my $fileCheck = compare($url, $compareFile);
104    if ($fileCheck != 0) {
105
106        # If a similar copy isn't found in the "csp" directory,
107        # it means that the file is being sent to the CSP, and is to
108        # be copied to the "csp" directory.
109        # Before sending the encrypted file to the "csp" directory,
110        # we need to verify that the two encrypted files produced by the
111        # ENC_MODULE and ENC_MODULE2 containers have been seen, and that
112        # they have the same content.
113        # This is done to ensure that none of the two encryption applications
114        # is attempting to leak data to the CSP, either by embedding it in the

```

```

115      # encrypted file , or via a covert channel where it pads the encrypted file
116      # to a certain size in order to transmit information , or arbitrarily sending
117      # files that were not sent by the user.
118      #
119      # We therefore stall for one minute , and then verify that a similar
120      # copy of the encrypted file that triggered this script exists in
121      # the "encrypted2" directory (which interfaces with the ENC_MODULE2
122      # container).
123
124      sleep(60);
125      my $verStatus = FileVerification($url , $fname , $encrypted2);
126      if($verStatus != 0) {
127
128          # if a similar copy of this file isn't found in the "encrypted2" directory ,
129          # we disable incron , which coordinates synchronization , so that no more file
130          # synchronization with the CSP will take place , until the user restarts
131          incron.
132          # We also log the event.
133
134          'kill -f incron ';
135          open(LOG, ">>$auditLog") || die("failed to open $auditLog\n");
136          print LOG "File check failed! Second copy of file $fname was not seen
137                  in $encrypted2!\n";
138                  close(LOG);
139      }
140  }
141  }
142  }
143  }
144  }
145  }
146  }
147  }
148  }
149  }
150  }
151  }
152  }
153  }
154  }
155  }
156  }
157  }
158  }
159  }
160  }
161  }
162  }
163  }
164  }
165  }
166  }
167  }
168  }
169  }
170  }
171  }
172  }
173  }
174  }
175  }
176  }
177  }
178  }
179  }
180  }
181  }
182  }
183  }
184  }
185  }
186  }
187  }
188  }
189  }
190  }
191  }
192  }
193  }
194  }
195  }
196  }
197  }
198  }
199  }
200  }
201  }
202  }
203  }
204  }
205  }
206  }
207  }
208  }
209  }
210  }
211  }
212  }
213  }
214  }
215  }
216  }
217  }
218  }
219  }
220  }
221  }
222  }
223  }
224  }
225  }
226  }
227  }
228  }
229  }
230  }
231  }
232  }
233  }
234  }
235  }
236  }
237  }
238  }
239  }
240  }
241  }
242  }
243  }
244  }
245  }
246  }
247  }
248  }
249  }
250  }
251  }
252  }
253  }
254  }
255  }
256  }
257  }
258  }
259  }
260  }
261  }
262  }
263  }
264  }
265  }
266  }
267  }
268  }
269  }
270  }
271  }
272  }
273  }
274  }
275  }
276  }
277  }
278  }
279  }
280  }
281  }
282  }
283  }
284  }
285  }
286  }
287  }
288  }
289  }
290  }
291  }
292  }
293  }
294  }
295  }
296  }
297  }
298  }
299  }
300  }
301  }
302  }
303  }
304  }
305  }
306  }
307  }
308  }
309  }
310  }
311  }
312  }
313  }
314  }
315  }
316  }
317  }
318  }
319  }
320  }
321  }
322  }
323  }
324  }
325  }
326  }
327  }
328  }
329  }
330  }
331  }
332  }
333  }
334  }
335  }
336  }
337  }
338  }
339  }
340  }
341  }
342  }
343  }
344  }
345  }
346  }
347  }
348  }
349  }
350  }
351  }
352  }
353  }
354  }
355  }
356  }
357  }
358  }
359  }
360  }
361  }
362  }
363  }
364  }
365  }
366  }
367  }
368  }
369  }
370  }
371  }
372  }
373  }
374  }
375  }
376  }
377  }
378  }
379  }
380  }
381  }
382  }
383  }
384  }
385  }
386  }
387  }
388  }
389  }
390  }
391  }
392  }
393  }
394  }
395  }
396  }
397  }
398  }
399  }
400  }
401  }
402  }
403  }
404  }
405  }
406  }
407  }
408  }
409  }
410  }
411  }
412  }
413  }
414  }
415  }
416  }
417  }
418  }
419  }
420  }
421  }
422  }
423  }
424  }
425  }
426  }
427  }
428  }
429  }
430  }
431  }
432  }
433  }
434  }
435  }
436  }
437  }
438  }
439  }
440  }
441  }
442  }
443  }
444  }
445  }
446  }
447  }
448  }
449  }
450  }
451  }
452  }
453  }
454  }
455  }
456  }
457  }
458  }
459  }
460  }
461  }
462  }
463  }
464  }
465  }
466  }
467  }
468  }
469  }
470  }
471  }
472  }
473  }
474  }
475  }
476  }
477  }
478  }
479  }
480  }
481  }
482  }
483  }
484  }
485  }
486  }
487  }
488  }
489  }
490  }
491  }
492  }
493  }
494  }
495  }
496  }
497  }
498  }
499  }
500  }
501  }
502  }
503  }
504  }
505  }
506  }
507  }
508  }
509  }
510  }
511  }
512  }
513  }
514  }
515  }
516  }
517  }
518  }
519  }
520  }
521  }
522  }
523  }
524  }
525  }
526  }
527  }
528  }
529  }
530  }
531  }
532  }
533  }
534  }
535  }
536  }
537  }
538  }
539  }
540  }
541  }
542  }
543  }
544  }
545  }
546  }
547  }
548  }
549  }
550  }
551  }
552  }
553  }
554  }
555  }
556  }
557  }
558  }
559  }
560  }
561  }
562  }
563  }
564  }
565  }
566  }
567  }
568  }
569  }
570  }
571  }
572  }
573  }
574  }
575  }
576  }
577  }
578  }
579  }
580  }
581  }
582  }
583  }
584  }
585  }
586  }
587  }
588  }
589  }
590  }
591  }
592  }
593  }
594  }
595  }
596  }
597  }
598  }
599  }
600  }
601  }
602  }
603  }
604  }
605  }
606  }
607  }
608  }
609  }
610  }
611  }
612  }
613  }
614  }
615  }
616  }
617  }
618  }
619  }
620  }
621  }
622  }
623  }
624  }
625  }
626  }
627  }
628  }
629  }
630  }
631  }
632  }
633  }
634  }
635  }
636  }
637  }
638  }
639  }
640  }
641  }
642  }
643  }
644  }
645  }
646  }
647  }
648  }
649  }
650  }
651  }
652  }
653  }
654  }
655  }
656  }
657  }
658  }
659  }
660  }
661  }
662  }
663  }
664  }
665  }
666  }
667  }
668  }
669  }
670  }
671  }
672  }
673  }
674  }
675  }
676  }
677  }
678  }
679  }
680  }
681  }
682  }
683  }
684  }
685  }
686  }
687  }
688  }
689  }
690  }
691  }
692  }
693  }
694  }
695  }
696  }
697  }
698  }
699  }
700  }
701  }
702  }
703  }
704  }
705  }
706  }
707  }
708  }
709  }
710  }
711  }
712  }
713  }
714  }
715  }
716  }
717  }
718  }
719  }
720  }
721  }
722  }
723  }
724  }
725  }
726  }
727  }
728  }
729  }
730  }
731  }
732  }
733  }
734  }
735  }
736  }
737  }
738  }
739  }
740  }
741  }
742  }
743  }
744  }
745  }
746  }
747  }
748  }
749  }
750  }
751  }
752  }
753  }
754  }
755  }
756  }
757  }
758  }
759  }
760  }
761  }
762  }
763  }
764  }
765  }
766  }
767  }
768  }
769  }
770  }
771  }
772  }
773  }
774  }
775  }
776  }
777  }
778  }
779  }
780  }
781  }
782  }
783  }
784  }
785  }
786  }
787  }
788  }
789  }
790  }
791  }
792  }
793  }
794  }
795  }
796  }
797  }
798  }
799  }
800  }
801  }
802  }
803  }
804  }
805  }
806  }
807  }
808  }
809  }
810  }
811  }
812  }
813  }
814  }
815  }
816  }
817  }
818  }
819  }
820  }
821  }
822  }
823  }
824  }
825  }
826  }
827  }
828  }
829  }
830  }
831  }
832  }
833  }
834  }
835  }
836  }
837  }
838  }
839  }
840  }
841  }
842  }
843  }
844  }
845  }
846  }
847  }
848  }
849  }
850  }
851  }
852  }
853  }
854  }
855  }
856  }
857  }
858  }
859  }
860  }
861  }
862  }
863  }
864  }
865  }
866  }
867  }
868  }
869  }
870  }
871  }
872  }
873  }
874  }
875  }
876  }
877  }
878  }
879  }
880  }
881  }
882  }
883  }
884  }
885  }
886  }
887  }
888  }
889  }
890  }
891  }
892  }
893  }
894  }
895  }
896  }
897  }
898  }
899  }
900  }
901  }
902  }
903  }
904  }
905  }
906  }
907  }
908  }
909  }
910  }
911  }
912  }
913  }
914  }
915  }
916  }
917  }
918  }
919  }
920  }
921  }
922  }
923  }
924  }
925  }
926  }
927  }
928  }
929  }
930  }
931  }
932  }
933  }
934  }
935  }
936  }
937  }
938  }
939  }
940  }
941  }
942  }
943  }
944  }
945  }
946  }
947  }
948  }
949  }
950  }
951  }
952  }
953  }
954  }
955  }
956  }
957  }
958  }
959  }
960  }
961  }
962  }
963  }
964  }
965  }
966  }
967  }
968  }
969  }
970  }
971  }
972  }
973  }
974  }
975  }
976  }
977  }
978  }
979  }
980  }
981  }
982  }
983  }
984  }
985  }
986  }
987  }
988  }
989  }
990  }
991  }
992  }
993  }
994  }
995  }
996  }
997  }
998  }
999  }
1000  }

```

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 that triggered
13 # this script.
14 # – The variable "encrypted" is assigned the full path of the directory
15 # that interfaces with the ENC_MODULE container
16 # – The variable "csp" is assigned the full path of the directory
17 # that interfaces with the user's directory (USER_DIR)
18 # – The "auditLog" variable is assigned the full path of the file we use
19 # 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 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 "encrypted" directory.
33 # It returns a status code indicating whether or not a similar file
34 # was found.
35
36     my $fileURL = $_[0];
37     my $fileName = $_[1];
38     my $dir = $_[2];
39     my $compareFile = $dir . $fileName;
40     my $compareStatus = compare($fileURL, $compareFile);
41     return $compareStatus;
42 }
43
44 sub Main {
45
46 # This script monitors the "encrypted2" directory in the VERIFICATION
47 # container, which interfaces with the encryption container ENC_MODULE2.
48 # When a file, to be synced to the CSP (created, or updated by the
49 # user), or which has just been received from the CSP,
50 # lands on this directory, this script is triggered, by the matching rule
51 # specified in the incrontab file (/var/spool/incron/root),
52 # and this matching rule passes the url of the file to this script.
53 #
54 # Note that the rules, specifying the directories to be monitored by incron,
55 # the events to monitor, and the actions to take when the events occur,
56 # are created during the configuration setup phase
```

```

57 # (counter-covertchannels_config.sh — specifically in the Aux_Image_Setup ()
58 # function)
59
60     chomp( $url );
61
62 # we extract the file name from the url
63
64     my $fname = fileparse($url);
65     my $compareFile = $csp . $fname;
66
67 # We check if the file that triggered the script has a similar copy
68 # in the "csp" directory. If a similar copy exists, it means
69 # that the file should not be sent to the "csp" directory, since it
70 # has just been received from the CSP, and is being synced to the user.
71 # This helps distinguish between the case where a file entering the
72 # "encrypted2" directory(which triggers this script), is from the CSP
73 # and needs to be decrypted before being sent to the user, or is from
74 # the user. We need to distinguish these two cases in order to avoid
75 # getting into an infinite loop.
76
77     my $fileCheck = compare($url, $compareFile);
78     if($fileCheck != 0) {
79
80         # If a similar copy isn't found in the "csp" directory,
81         # it means that the file is being sent to the CSP.
82         # We need to verify that a similar copy of this file was received in the
83         # "encrypted" directory (which interfaces with the ENC_MODULE container).
84         #
85         # This is done to ensure that none of the two encryption applications
86         # is attempting to leak data to the CSP, either by embedding it in the
87         # encrypted file, or via a covert channel where it pads the encrypted file
88         # to a certain size in order to transmit information, or even sending files
89         # that were not sent by the user.
90         # We also need to prevent a malicious encryption application from being able
91         # to arbitrarily send the encrypted file, and thus leak data to the colluding
92         # by modulating the sending time.
93         #
94         # We therefore stall for one minute, and then verify that a similar
95         # copy of the encrypted file that triggered this script exists in
96         # the "encrypted" directory (which interfaces with the ENC_MODULE
97         # container).
98         # If we find a similar copy of the file in the "encrypted" directory,
99         # we do nothing and exit.
100
101         sleep(60);
102         my $verStatus = FileVerification($url, $fname, $encrypted);
103         if($verStatus != 0) {
104
105             # if a similar copy of this file isn't found in the "encrypted"
106             # directory,
107             # we disable incron, which coordinates synchronization, so that no
108             # more file
109             # synchronization with the CSP will take place, until the user
110             # restarts incron.
111             # We also log this event.

```

‘pkill -f incrond’;

```
111         open(LOG, ">>$auditLog") || die("failed to open $auditLog\n")
112         ;
113         print LOG "File check failed! Second copy of file $fname was
114         not seen in $encrypted!\n";
115         close(
116         LOG);
117     }
118 }
119
120 Main() ;
```