Using the Certifier Framework for Confidential Computing

Major Concepts

The Certifier Framework for Confidential Computing consists of two major software elements:

- The Certifier API, which is a small API designed to allow you to use Confidential Computing with a minimum of effort.
- The Certifier Service, which allows Confidential Computing programs to be deployed and managed in a simple, scalable manner.

Confidential Computing relies on isolation of Confidential Computing programs from all other programs, the unforgeable identity of application elements (measurements), secure key management (using Seal/Unseal) and rigorous verification of other Confidential Computing programs under a specific machine enforced policy as a basis for collaboration (Attestation and policy control).

The foundation for Confidential Computing is complete knowledge of each Confidential Computing program in a security domain. Programs can only act in accordance with verified program properties. In addition, trust decisions are rooted in a policy key, in the control of the security domain owner. All program decisions related to what hardware and programs to trust is rooted in the policy key. The policy key signs approved policy and only verified policy signed (or policy key delegated) policy is used in trust or access decisions.

> ### Four capabilities of a Confidential Computing:
>
> - **Isolation.** Program address space and computation.
> - **Measurement.** Use cryptographic hash to create an unforgeable program identity.
> - **Secrets.** Isolated storage and exclusive program access. (aka, "sealed storage").
> - **Attestation.** Enable remote verification of program integrity and secure communication with other such programs.

Confidential Computing Properties at a glance

This means that secure hardware and securely written Confidential Computing programs are unconditionally protected. Confidential Computing provides a principled, verifiable security mechanism for distributed computing (across the multi-cloud!); it protects the integrity and confidentiality of processing **wherever** programs run from other malicious programs or even malicious insiders (e.g., admins).
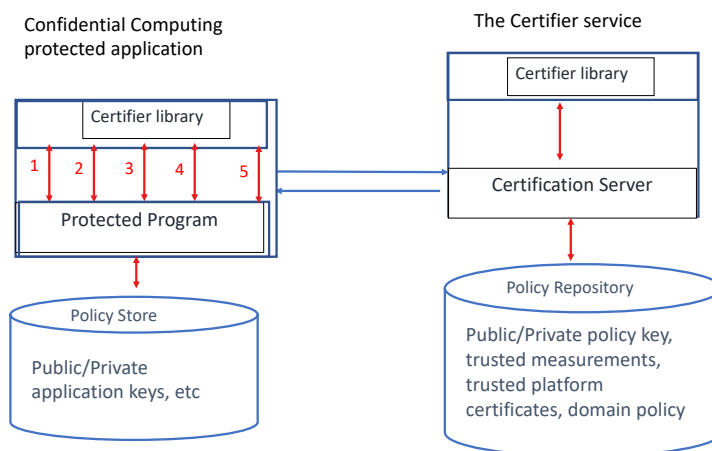
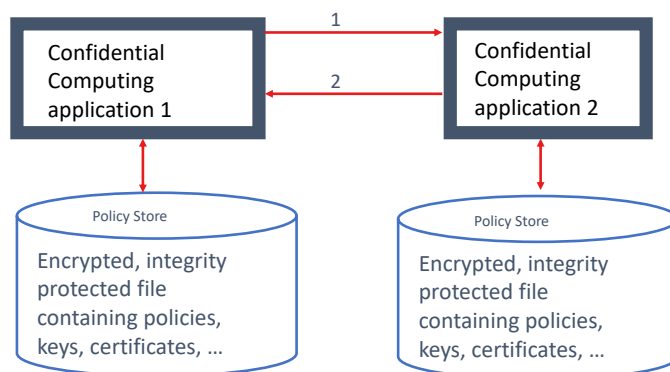Figure 1: Certifier API and Certifier Service

Figure 2: Two mutually authenticated Confidential Computing Applications communicating over a secure channel. The thick box indicates isolation.

Environments

The certifier runs identically in different environments.

Firstly, it can use SGX, SEV and in the future other "hardware enforcement" mechanisms supporting the Isolation, Measurement, Sealing, Unsealing and Attestation without modifying the program. In addition, the certifier comes with a "simulated-enclave" so you can develop

and test on platforms without special hardware. Many of the sample applications us it to illustrate the API.

On "enforcing hardware," a client can run in the following ways:

1. In a "simulated-enclave," which can run without CC hardware. This allows debugging and testing.
2. In an SGX enclave under Open Enclaves, Asylo, or Gramine. Here the application enclave is the isolated a measured application enclave.
3. In a "application service enclave" in an encrypted virtual machine under SEV or TDX. Here the entire kernel and initramfs is the isolated and measured security principle. The "application service" provides service for OS-wide actions.
4. In an application (process) in an encrypted virtual machine under SEV or TDX.   Here, the process is the isolated and measured principal. An OS-wide "application service" provides Isolation, Measurement, Sealing, Unsealing and Attestation services for client application. The application service itself is protected by the secure encrypted virtual machine platform and provides Confidential Computing services (using its own protected keys) for client programs isolated by OS level process isolation. All such protected programs are descendants (e.g-children) of the "application service" and enjoy the same certifier API and certification as programs protected at the platform level.   In this case, the application service enforces trust policy using attestation and other services performed by the hardware and the process level attestation performed by an application program enforces trust policy securely provided by the `application_service`. Notes on using the application service and writing applications that run under it are in the `application_service` directory. An example, using the `application_service` is in `sample_apps/simple_app_under_app_service`.

The certifier API and Certifier Service works the same way in each of these cases. We refer to any of these Confidential Computing protected programs as "applications" below without distinction.

Writing Applications

The certifier API makes converting a well written application into a Confidential Computing enabled application easy.   There are sample applications in the `sample_apps` directory. Each comes with complete instructions.

The certifier performs a number of functions:

1. It abstracts the underlying isolate, measure, seal, unseal and attest primitives so they have the same interface in any environment.

2. It provides a secure store which can be securely saved and recovered (in one statement!). The store will contain keys, public keys for authentication, policies, symmetric keys for encrypting and integrity protecting files and certificates and tokens acquired by the program to carry out its functions.
3. It provides a policy language, evidence formats and policy evaluation to help a Confidential Computing application determine when another Confidential Computing application should be trusted according to signed policy. Evidence submitted and evaluated includes attestation reports from the platform(s).
4. It contains a mechanism to establish secure channel (encrypted, integrity protected bi-directional channels with authenticated trusted enclave named by their measurements).
5. It contains "helpers" APIs, for example file encryption, file protection using application file keys (so one needn't decrypt files to transfer them to another Confidential Computing application), policy language manipulation, human readable proofs of trust decisions.
6. Mechanisms to establish trust bilaterally between two Confidential Computing applications and, more usefully a mechanism for a Confidential Computing application to prove its trustworthiness within a security domain (defined by policy) to the Certifier Service which provides a one-stop "admission certificate" to establish trust thereafter with any Confidential Computing application in the security domain. This mechanism allows for scalable applications with the ability to upgrade without redistributing application.
7. Utilities to generate keys and write policy.

The process of modifying a well written application to make it a Confidential Computing application is rather thoroughly illustrated in the "simple_app" in the `sample_apps` directory. In `simple_app`, a single executable compiled from `example_app.cc` acts both as a Confidential Computing application client (from the point of view of TLS) and as a server. The Certifier Service runs on one or more servers. The Certifier Service evaluates, or certifies, all the policy for the applications that run in a security domain (`example_app.cc`, in this case). This evaluation results in an "Admission Certificate" within the security domain (a domain complying with policy identified by a "policy key"), applications need know nothing about the policy details; indeed, the applications can run securely in any properly configured security domain even as policy changes.

Here are some comments about the process.

1. A Confidential Computing application must have an associated policy key which is a public/private key pair. If a Confidential Computing application wishes to use the Certifier Service, the Certifier generates the key-pair (i.e.- the policy key) and a self-signed cert for the key. The private portion of the policy key is used only by the Certifier Service.
2. If a Confidential Computing application wishes to use the Certifier Service, the policy key, which roots all decisions must be embedded in the applications. As described, in `sample_apps/simple_app/policy_key_notes.txt`, there are three ways to do this,

but the easiest involves providing the Certifier Service provisioned self-signed policy key certificate to a utility (`embed_policy_key.exe`) which puts it in the `example_app.exe` application. In this case, the policy key is part of the measurement of the Confidential Computing application.

3. When a Confidential Computing application starts for the first time it generates an authentication key, called an enclave-key, which allows it to authenticate itself to other programs in the security domain and puts the private and public portion of the enclave-key in the policy store. This is done in `cold_init` in `cc_helpers.cc`. That routine also generates some (optional) symmetric keys that can be used to encrypt, and integrity protect files; those keys are also stored in the policy store. In fact, many support routines for Confidential Computing, can be found in `cc_helpers.cc`.

4. Next, `example_app.cc`, requests an attestation, naming the enclave public key and recovers some additional evidence supporting a trust decision from the platform. It assembles these into a `trust_request_message` and sends it to the Certifier Service. The Certifier Service evaluates the evidence in conjunction with security domain policy and sends back a `trust_response_message` indicating the evaluation was successful and including an "admission certificate" which names the Confidential Computing application measurement and its public enclave-key. This is done in the routine `certify_me()` in `cc_helpers.cc`, this routine is called in `example_app.cc` to illustrate its use.

5. The Confidential Computing application extracts the admission certificate from the response and stores it in the policy store. It saves the policy store so all the information in it can be retrieved whenever the program restarts. The policy store is automatically encrypted, and integrity protected with keys that are sealed to `example_app.exe` on the platform.

6. At this point, `example_app.exe` can either continue or restart later. If it decides to restart later, it recovers its policy store (which is decrypted, and integrity checked by the certifier API) and retrieves its enclave public/private key pair as well as its admission certificate. This is done in `warm_restart()`.

7. At this point, the Confidential Computing application, `example_app.exe`, is fully initialized and proceeds with normal processing. If it wishes to contact another Confidential Computing application in the security domain (in this case, an instance of the very same `example_app.exe` on another machine), it uses its enclave key and admission certificate to open a bidirectional channel with the other Confidential Computing application (which symmetricly uses its enclave key and admission certificate in the channel negotiation). The client side of this is performed by one of the participants using `client_auth_client(SSL* ssl)` and the server side is performed by the other participant using `server_application(SSL* ssl)`. Very likely, you can copy and use all these routines in your program with few changes.

8. During execution, a Confidential Computing application may transmit secrets or data to another trusted Confidential Computing application within the security domain using this secure channel. When it does so, it knows that only the identified (by its measurement) authenticated program complying with the domain security policy can get it.

9. The Confidential Computing application, during execution, may also obtain keys to shared distributed files or wish to securely write, or read previously securely written files and the certifier provides a one-step way to do that as well as other common functions.

That's it! You can now imagine converting any program or service into on protected by Confidential Computing rather quickly with the certifier API.

The code in `sample_apps/simple_app` and accompanying instructions provide a complete step by step guide to writing Confidential Computing Applications and deploying them.

There is a more extensive sample machine learning enclave that analyzes data; this example is in `simple_apps/analytics_example`. There is also a sample encrypted virtual machine service in `simple_apps/att_md_service`. In addition, there are code variants of simple_app as follows:

1. In the directory `simple_apps/simple_app_under_sev`, is the same simple_app running in AMD-SEV.
2. In the directory `simple_apps/simple_app_under_oe`, is the same simple_app running in SGX using the Open Enclaves SDK.
3. In the directory `simple_apps/simple_app_under_application_service`, is the same simple_app running under a system service provided either by the simulated_enclave, or SEV (or TDX in the future).
4. There is an asylo based example in `simple_apps/asylo_secure_grpc` that uses the ported Asylo SDK.

Each of the above examples come with rather complete instructions that include deployment examples, policy generation and tests as well as "platform specific" notes (like how to assemble ab SEV capable VM). In general, we will add a "simple_app" example for each platform we support to simplify use. The "sample_app" shows just how little changes on different platforms when you use the certifier. By the way, contributed applications are welcome!

Configuring Policy and running the Certifier Service

Policy is expressed in a declarative policy language rooted in a policy key. The Confidential Computing Framework provides tools to author, read, and distribute policy. Confidential Computing programs that other Confidential Computing programs wish to rely on (trust) must first submit evidence (including attestations) to establish their trustworthiness as well as have an unforgeable way to authenticate themselves. This trust decision is made in a mathematically rigorous way using only the evidence, policy, and logic.

This process is described in detail in the simple sample application provided with this repository in `sample_apps/simple_app/instructions.txt`; there is also a helpful script in that directory.

Proofs from the Certifier Service

Trust decisions are accompanied by short, human readable proof. We have an internal evidence and policy format based on the Lampson-Abadi SPKI/SDSI formalism with constrained delegation. The internal format consists of simple predicates with key or measurement-based principals; these statements are called "`vse-clauses`." You can also use other claim formats (like certs) or substitute another policy evaluation engine, like Datalog or OPA as the indicated in the code.

The internal evidence format has the advantage that it is simple and easily read (by humans!). We haven't come across a policy we can't express rapidly in this format. Policy is produced by the policy tools in the utilities directory. Consult `sample_apps/simple_app/intructions.txt` for further information.

Here is an example "proof" that uses policy and application provided evidence (including an attestation). Remember, the goal is to prove the enclave-key can be trusted for authentication within the security domain based on policy and evidence.

Proof

```
1. Key[rsa, policyKey, c9d16649…] is-trusted
   and
   Key[rsa, policyKey, c9d1664…] says Measurement[cdf3590…]is-trusted
   imply via rule 3
   Measurement[cdf35…] is-trusted

2. Key[rsa, policyKey, c9d16649…] is-trusted
   and
   Key[rsa, policyKey, c9d16649…2] says Key[rsa, platformKey, e59709…]
   is-trusted-for-attestation
   imply via rule 5
   Key[rsa, platformKey, e59709bae…] is-trusted-for-attestation

3. Key[rsa, platformKey, e59709bae…] is-trusted-for-attestation
   and
   Key[rsa, platformKey, e59709bae4…] says Key[rsa, attestKey,
   e3f0bbd20a…] is-trusted-for-attestation
   imply via rule 5
   Key[rsa, attestKey, e3f0bbd2…] is-trusted-for-attestation

4. Key[rsa, attestKey, e3f0bbd2…] is-trusted-for-attestation
```

```
and
Key[rsa, attestKey, e3f0bbd2…] says Key[rsa, app-auth-key,
 b86447b…] speaks-for Measurement[cdf359…1]
imply via rule 6
Key[rsa, app-auth-key, b86447b71e…] speaks-for
Measurement[cdf359089b4…]
```

```
5. Measurement[cdf3590…1] is-trusted
   and
   Key[rsa, app-auth-key, b86447b71e…] speaks-for
   Measurement[cdf35908…]
   imply via rule 1
   Key[rsa, app-auth-key, b86447b7…] is-trusted-for-authentication
```

The conclusion of step 5 (in bold) is what we were after.

Notes and observations

1. The Certificate Service and Certifier API are format rule agnostic. Any tokens or formats you use in an application or service work the same way they used to. No need for token translation or a change in application authorization logic.
2. Provisioning of keys and data requires almost no change to existing applications. Basic keys are either generated by the application or transmitted via a secure channel from a trusted application in the security domain. Data is provisioned as before, except through a secure channel.
3. The certifier does not rely on root key store, application actions are entirely controlled by signed policy from the policy key.
4. Neither the Certifier nor the Certifier Service requires any changes in application provisioning or deployment. Any existing mechanism continues to work.
5. Confidential Computing applications can be written in C, C++ or Go and via shims all the other popular languages.
6. The Certifier Service can add or upgrade individual Confidential Computing applications without redeploying exiting ones.
7. The entity controlling the Certifier Service is in complete control of the security domain. No action can be taken, no data can be changed, modified, or read unless it conforms to policy. You can run the Certifier Service yourself (with minimal overhead and resilience and availability) consuming minor server resource or you can have someone run it on your behalf.
8. Admission to the security domain relies on a trust decision (usually supported by code inspection) of applications "admitted" to the security domain. Confidentiality and integrity of processing depends only on the Confidential Computing applications (which you either wrote or had an opportunity to review in its entirety or had a third party do so) and hardware enforcement. There is no dependency on third parties or service providers for these properties. Neither improper configuration within a service provider

(or on your own machines!), nor malicious administrators, nor malware can compromise your Confidential Computing applications.

9. You can use this framework for collaborative Confidential Computing workloads without disclosing data to other participants.
10. When programming a Confidential Computing program in an encrypted virtual machine, ordinary Linux service calls work in a manner that programmers are familiar with so no additional training is required for programmers who know how to write secure applications. When programming in an SGX enclave, platform calls are provided by an SDK like Open Enclaves or Gramine.
11. There is an end-to-end Open Enclaves test (which includes Open Enclaves instructions) in openenclaves_test.
12. Other token formats can be issued by the Certifier Service, although the X509 certificate, which is used to open mutually authenticated channel between "certified" confidential computing programs, is probably the most universally useful.
13. The Certifier API code comes with a gtest based set of tests and there is gtest based certifier tests in `certifier_service/certlib`. There are many "standalone" tests; among these is "test_secure_channel" for testing secure channels, test_size_client/server to test channels, pipe_read_test to test service to application API channels in encrypted virtual machines, and each application (like simple_app) comes with complete end-to-end tests.

Some Applications

Here are some applications, several of which we have implemented to make sure the Certifier Framework for Confidential Computing is easy (and safe) to use:

1. Hardware secure module
2. Secure key store and token generation
3. Secure motion planning as a service
4. Secure collaborative machine learning
5. Secure auctions
6. Secure real-time trading services
7. Secure Kubernetes container management (via secure Spiffie/Spire)
8. Secure federated identity management
9. Secure databases
10. gRpc
11. Secure document sharing
12. Secure sensor collection
13. Secure caching services
14. Standard platform components (storage, logging, time, IAM)

Advice

We strongly recommend following the instructions and reviewing the code in `sample_apps/simple_app` which gives a complete picture of all aspects of using the Certifier API and Certifier Service.

Using the Certifier Framework for Confidential Computing

Suggestions and contributions are warmly welcomed. The repository is at github.com/vmware-research/certifier-framework-for-confidential-computing

Appendix --- API

The API is specified in three include files, `support.h,` `certifier.h` and `cc_helpers.h` in the include directory.

In addition, there is an automatically generated header file for the protobufs and you will find the protobuf definitions in `src/certifier.proto`. You will likely use only a few of these calls in any application, namely, the Confidential Computing Primitives and the Policy Store and these are all illustrated in the example code. You may also want to use some of the "helper" routines in `sample_apps/simple_app/example_app.cc`; these should help you get you apps working quickly. You may also want to use the procedures in these samples to run prototype Certifier Service instances. Have fun!

Tour of simple_example and other apps

We have emphasized the value of the simple applications in `sample_apps` as a way to understand how to write apps and deploy and manage them using the Certifier Framework. Here we provide a tour to help understand those examples.

The principal interfaces for the Certifier API are in the files support.cc and certifier.cc. However, most of the patterns used in writing an application are very similar so we also supply several common helper functions in cc_helper.cc. There are several different "enclaves" supported in the Certifier Framework (and we'll be adding more). These include:

1. A simulated enclave which allows you to develop on any machine by providing a simulated Confidential Computing environment. The example here is in the directory `sample_apps/simple_app`.
2. An SEV-SNP based enclave which employs SEV-SNP (at the VM level) to provide a Confidential Computing environment. The example here is in the directory `sample_apps/simple_app_under_snp`.
3. An SGX based enclave which employs SGX to provide a Confidential Computing environment.
4. An application service-based enclave which provides a Confidential Computing environment to applications within an encrypted virtual machine (like an SEV-SNP machine). This service is rooted and SEV-SNP service like the one in 2 above. The example here is in the directory `sample_apps/simple_app_under_app_service`.

We implement the simple example in each of these to demonstrate the (rather small) differences in using each of these environments as well as the commonality achieved by using the Certifier Framework in each of these environments. In the text below, we focus on describing the code itself.

Simple_app guide

The procedures for building the app and running the certifier service is detailed in the file `sample_apps/simple_app/structions.txt`. In this example, we actually compile all the certifier files along with the app using the make file `example_app.mak`. Although there is only one application binary, `example_app.exe`, the binary serves the role of two enclaves, one acting as a server and one as a client; a flag selects which one is being used in an invocation.

Below we provide a detailed description of *all* the steps carried out by a developer or a deployer of a Confidential Computing based application ecosystem. The simple app uses a simulated environment called a "simulated-enclave." Running under different environments (SGX, SEV-SNP or TDX) requires only a parameter change in the calls.

Referring to the enumerated steps in `instructions.txt`:

- Step 1 involves compiling utility programs used to initialize keys and write policy for the Certifier Service we'll be running.
- In step 2, we create a directory for application data provisioning.
- In step 3, we create the enclave and application data. For the simulated enclave, this includes the policy key, the self-signed policy key cert as well as keys used by the simulated enclave to provide the Confidential Computing primitives like Seal, Unseal and Attest.
- In step 4, we generate a file that contains the self-signed policy key certificate that will be embedded in the application `example_app.exe`.
- Having produced the files needed, we compile the application `example_app.exe` in step 5.
- In step 6, we use the utility `measurement_utility.exe` to measure `example_app.exe`. We need the measurement to write policy for the Certifier Service.   For SEV-SNP, Open Enclaves (using SGX) and Asylo (using SGX) we will use different utilities to produce the measurements.
- Step 7 constructs all the policy statements (in our policy language) that must be provisioned to the Certifier Service as well as a "platform key" rule which would be supplied by the platform provider (e.g.- Intel or AMD) in hardware backed enclaves. The policy for the policy is bundled into the file `policy.bin`, which will be provided to the Certifier Service. It is a very simple policy consisting of just two statements which, in short-hand are:
    1. "The policy-key says the measurement (the one we calculated in step 6) is-trusted."
    2. "The policy-key says the platform-key (a key for the class of hardware, like SEV-SNP) is-trusted-for-attestation."

    The platform-key rule which is usually obtained on the hardware platform (but we must construct it for the simulated enclave) is:

    The platform-key (in number 2 above) says the attestation-key (the one on the hardware we are using, in the simulated enclave case, a key we generated in step 3) is-trusted-for-attestation.

- In step 8, we compile the Certifier Service in Go.
- In step 9, we copy the needed data files into the subdirectories for the Certifier Service (service) as well as each of the app roles for `example_app.exe`. These directories are created in steps 10, 11 and 12.
- In a new window, in step 13, we run the Certifier Service which reads in policy rules provisioned in step 9.
- In step 14, we invoke the application `example_app.exe` in each of its roles (an SSL client and an SSL server) to initialize its keys, and contact the Certifier Service in each case,

offering proof of its compliance with the policy domain policy (created from rules an attestation generated in the application). All this is stored in the policy store which is then securely saved after being encrypted and integrity protected.

- In step 15, in different windows, corresponding to each of the `example_app.exe` roles, we first run the application acting as an SSL server, in the one window and then, in another window, run the application acting as an SSL server. The server will send the client a message "Hello from your secret server" and the client will send the server the message "Hello from your secret client" over a secure channel rooted in the policy key using the "Admission" certificates the application obtained (for each role) from the Certifier Service.

The certifier uses a helper class called `secure_authenticated_channel`, which liberates programmers from needing to know the openssl (or boringssl) calls required to implement a mutually authenticated, encrypted, integrity protected, channel with another trusted application in the security domain. In addition, `secure_authenticated_channel` keeps track of the measurement of the peer connection for both the client and the server. This makes implementing ACLs for granular access control very easy. It also cuts down on the code a developer needs to add for almost all application to turn an application into a "Confidential Computing" protected application. Other helpers also perform most of the routine tasks of generating keys, verifying rules and storing information securely.

Using the helpers, the developer usually adds only a few calls to initialize the environment and carry out verification. In fact, as simple example illustrates, this usually involves the following calls:

```
1. app_trust_data->cold_init(public_key_alg,symmetric_key_alg,
   hash_alg, hmac_alg)
2. app_trust_data->certify_me(FLAGS_policy_host, FLAGS_policy_port)
3. client_application(secure_authenticated_channel& channel)
4. server_dispatch(host_name, port, asn1_policy_cert,
   private_key,private_key_cert, server_application)
```

The first call generates, and stores required security data. The second call contacts the Certifier Service to verify evidence, including an attestation the helpers generate, proving the application is "trustworthy" under the security enumerated policy (It is worth pointing out that this verification generates a proof and verification certificate, naming the application measurement.). The third call establishes an authenticated, secure channel with a proven trustworthy application on behalf of a client and the fourth call provides the corresponding secure channel establishment on behalf of the application the client wishes to contact. The first two calls are only required when the application first runs on a new platform. When it restarts, a single call:

- `app_trust_data->warm_restart()`

recovers the keys and certificates previously established. After trust establishment, simple application demonstrates sending data between two verified applications, namely, "Hi from your secret client" and "Hi from your secret server." Other than these steps, the developer need only encrypt, and integrity protect data that is stored locally or remotely either by using simple certifier calls or using secure storage services based on keys protected by the certifier.

Now, we'll highlight the application flow in `example_app.exe` using the Certifier Framework primitives.

- The "helper" object is created (`app_trust_data = new cc_trust_data(enclave_type, purpose, store_file);`). "store_file" is where we will save the policy store. The enclave type, in this case, is "simulated-enclave." The purpose is authentication. This last statement requires a little explanation. The helper object can serve in one of two roles. Most of the time the role is "authentication" where the application wishes to certify the public key it generates to authenticate the application. In the application service, the role is "attestation" where the application wishes to certify the public key it generates to provide attestation services to its children.
- Next, we retrieve the policy key (in a form that can be used to verify claims) from the embedded self-signed policy cert using the helper function `init_policy_key`.
- The next several lines of code are particular to the simulated enclave. Here we first construct the file names containing the keys and certificates used by the simulated enclave. Then, we supply those file names to a simulated specific initialization function in the helper (`initialize_simulated_enclave_data`). Every Confidential Computing provider will have a corresponding initialization function in the helper object. This is the only call that is different for different providers.
- For clarity, the app implements each of the common functions in different helper routines depending on the supplied operation flag.
    - If the operation is `cold-init`, the application generates all its keys (its authentication key, named in the Admission Certificate as well as symmetric keys it will use to encrypt, and integrity protect files)
    - If the operation is `warm-restart`, the application will retrieve the policy store and get its keys.
    - If the operation is `certify-me`, the application will construct evidence, which it provides to the Certifier Service, to certify its authentication key. It saves the resulting Admissions Certificate in the policy store for later use. These first three steps are performed by the app acting in each role.
    - If the operation is `run-app-as-server`, the application runs the `run_me_as_server routine` in the file to establish a mutually authenticated, encrypted, integrity protected SSL channel with clients that may contact it. This routine uses the authentication key, policy-key and Admissions certificate produced in earlier steps to establish the channel. When a client successfully opens such a channel, the application will know the certified authentication key and measurement of the (client) application which successfully opened the channel.

- o If the operation is `run-app-as-client`, the application runs the `run_me_as_client routine` in the file to establish a mutually authenticated, encrypted, integrity protected SSL channel with a server it wishes to contact. This routine uses the authentication key, policy-key and Admissions certificate produced in earlier steps to establish the channel. When a client successfully opens such a channel, the application will know the certified authentication key and measurement of the (server) application which successfully opened the channel.

Most of the other routines in example_app.cc are helpers to implement a mutually authenticated, encrypted, integrity protected, SSL channel with other applications in the security domain of the policy key. Note that the channel negotiation is rooted in the policy key (not keys from a root key store) using the Admissions Certificate.

That's it! Using these examples, a programmer can port existing applications rather quickly or produce new applications securely with very little additional effort. The Certifier Service can manage a collection of such secure programs scalably.

When using other enclaves (like SGX, SEV-SNP or the application-service enclave), the only difference involves a different "initialize" function in the third step above. Although this is the only difference, we actually implement simple_app for each provider (in `simple_app_under_sev` for SEV-SNP, and in `simple_app_under_app_service` for the app-service) to fully demonstrate platform dependent differences.

We used the helper object in cc_helper.cc, which provides an interface to rest of the Certifier API as required for almost all applications. Of course, you can access the functions in support.cc and certifier.cc on the rare occasions you need to do so but our goal is to minimize the calls to the certifier required in almost all applications.

The `analytics_example` uses these same steps to provide an Open Enclaves based data analysis application and the `asylo_secure-grpc` uses a protected Asylo based application to provide secured remote procedure calls.