Using the Certifier Framework for Confidential Computing

John, Ye and Aditya

Major Concepts

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

- The Certifier API library, which uses a small API designed to allow you to use Confidential Computing with a minimum of effort.
- The Certifier Service, a 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).

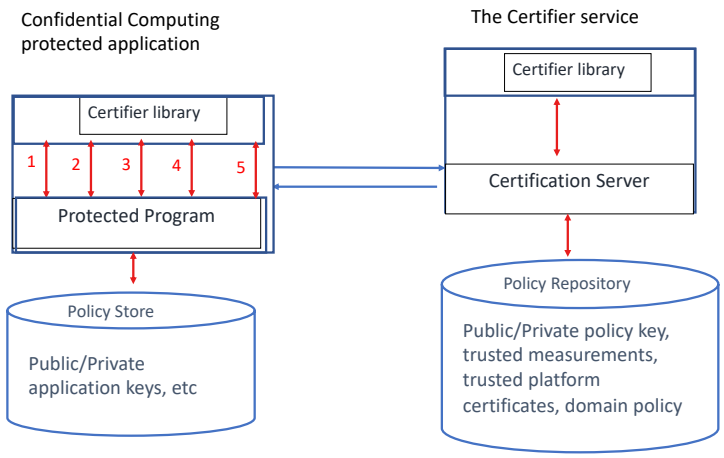

Figure 1: Certifier API and Certifier Service

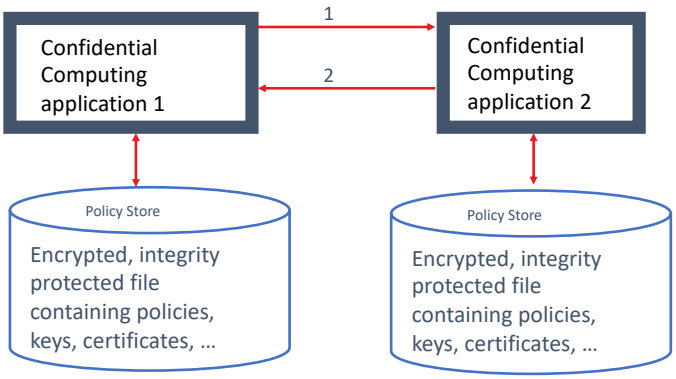

Figure 2: Two mutually authenticated Confidential Computing Applications communicating over a secure channel. The thick box indicates isolation.

Environments

The certifier runs identically in different environments: it can use SGX (under Open Enclaves or Gramine), 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. There is also support for an application service within an encrypted virtual machine which provides the Certifier API to individual applications within the VM.

This means a client can run:

1. In a "simulated-enclave," which can run without CC hardware. This allows debugging and testing.
2. In an SGX enclave under Open Enclaves or Gramine. Here the application enclave is the isolated a measured application enclave. We have done an Asylo port in the past, but we do not maintain it.
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[1]. 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

[1] The initial version of TDX is apparently inadequate as a Confidential Computing platform since it fails to provide Seal/Unseal or enabling enclave dependent key generation. Hopefully, subsequent versions of TDX will remedy this.

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 (whether an entire VM or a Unix process) as "applications" below without distinction.

Writing Applications

The certifier API makes converting a well written application into a Confidential Computing enabled application easy or, for that matter, writing a confidential computing protected program from scratch. There are sample applications in the `sample_apps` directory. Each comes with complete instructions. Many can be "copied and pasted" to get you started.

The certifier performs several 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.

Using the Certifier Framework API and its companion Certifier Service 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 symmetrically 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 using the simulated_enclave. Incidentally, the self-same `simple_app` example is also provided in `sample_apps` for SGX (under Open Enclaves and Gramine, SEV, the `application_enclave`, ARM and Keystone); you'll notice the application code itself hardly changes at all no matter what the protecting platform is.

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`.

There are code variants of `simple_app`, depending on the platform in `sample_apps` 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.
4. In the directory `simple_apps/simple_app_under_gramine`, is the same `simple_app` running in SGX using Gramine.
5. There are two additional platforms under construction that you may see in the repository, when they are working, we'll update this documentation.

Each of the above examples come with rather complete instructions, suitable for "copying and pasting" that include deployment examples, policy generation and tests as well as "platform specific" notes (like how to assemble an SEV capable VM). In general, we will add a "`simple_app`" example for each platform we support to simplify use. 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. In the case of SEV, we provide complete hardware verification (e.g. – firmware versions, revision levels, whether migration, debugging or key sharing in allowed). For Open Enclaves and Gramine, platform verification is performed by the SDK (and can be performed by external service) but we are planning to all complete SDK independent support for Gramine that works analogously to SEV.

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 and the `sample_apps` directory contains a rather handy way to compile and run `simple_app` on any of the supported hardware.

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 claim format consists of semantically clear, simple predicates with key or measurement-based principals. 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 yet come across a policy we can't express rapidly in this format. Policy is produced by the policy tools in the utilities directory or a policy generator that uses a json like syntax. Consult `sample_apps/simple_app/intructions.txt` for further information (or the policy_generator or run_examples.sh).

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 keys and data requires almost little 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 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.
14. The code targets Linux at present but most of the code runs on a MAC using the simulated-enclave which can be used for development but there are exceptions.

Consult the "Certifier Framework appendix" for more information about the Certifier API and how to use it.

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)

Multi-domain support

In August, 2023, we made changes to the Certifier API to support multi-domain programs. This requires minor changes to earlier versions of existing programs like the simple_app family of programs.

Recall that every program has an identified (usually embedded) policy key that specifiesthe security domain the program runs in. There is a certifier service associated with that security domain, namely the one that signs admissions certificates with that policy key.

Now programs can certify not only in their primary domain (the one associated with the embedded policy key) but in other domains with other policy keys. As a result, the principal object that carries out the Confidential Computing operations, has changed.

For existing programs that operate in a single security domain, like the simple_app family of examples, the only changes are:

1. The call to `cold_init` in `cc_trust_manager` has changed and takes more arguments. It is now bool cold_init(const string& public_key_alg, const string& symmetric_key_alg, const string& home_domain_name, const string& home_host, int home_port, const string& service_host, int service_port);

   public_key_alg is the public key algorithm used, as before.
   symmetric_key_alg is the symmetric key algorithm, as before.
   home_domain_name is the name of the home (primary domain) which identifies the domain. It can be any string as long as it is unique although it would be nice if it were descriptive.
   home_host is the ip address of the home domain's certifier service.
   home_port is the port number for the certifier service.
   service_host is the ip address of this applications server (in simple_app, this is the ip address of the ssl server exposed when the app runs as a server.
   service_port is the port number for the service.

All these variables were used in the earlier simple_app but initialized in different places. As always, the simple_app code completely illustrates its use.

2. The call certify_me() now takes no arguments (They are set as a consequence of cold_init now.

Those are the only programmer visible changes for simple-app like applications. None of the deployment instructions or procedures change at all.

You can now get sertified in secondary domains with different policy keys too. To do this, first register the new domain by calling:

bool add_new_domain(const string& domain_name, const string& cert, const string& host, int port, const string& service_host, int service_port);

Here,

domain_name is the secondary domain's domain name

cert is the der encoded self-signed policy key certificate for the secondary domain.
host is the ip address of the secondary domain's certifier service.
port is the port number for the secondary domains certifier service.
service_host and service_port have the same meaning as above.

Then call

```
bool certify_secondary_domain(const string& domain_name);
```

to have the program certified in the secondary domain. When you open a secure_authenticated_channel to a secondary domain, you should use the self-signed policy cert and admissions_cert for the *targeted* domain. multi_domain_simple_example is a complete example demonstrating these new calls.

These changes cause related changes in the key rotation examples.

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 --- Certifier Framework API

The public API for the Certifier API is specified in two include files: `certifier_framework.h` and `certifier_utilities.h` in the `include` directory. Studying the examples is the best way to understand the API but below we provide some guidance. In addition, the underlying API's are accompanied by tests in `certifier_tests` which can also serve as a useful reference. When you write an application, you will include these two ".h" files. Note that these API's are in the `certifier::framework` and `certifier::utilities` namespace, we omit these prefixes below.

Here are a few notes on the code style generally employed.

Syntactically, we use the Google C++ style guide. Generally, we employ Google protobufs for publicly facing data. Protobufs can be easily serialized and are extensible, but they avoid "generic" XML or JSON parsing pitfalls.

Input arguments to functions come first (left to right) in function definitions and are often const; output arguments come last and are pointers. Generally, output data structures are created by the caller.

We employ Google style byte serialization, that is: strings can hold binary values that are assigned and retrieved with standard string functions; the advantage is the string class manages data allocation.

The main interface for applications is in `certifier_framework.h;` it is organized around three classes:

> `class cc_trust_manager`: This class manages your keys, certificates and interacts with the certifier service to "certify" or recertify your application and obtaining an "admissions certificate" for it. You can also retrieve the policy key for your security domain from this class. The principal user accessible calls (all illustrated in the examples) are:
>
> The constructor and destructor, namely, `cc_trust_manager(const string& enclave_type, const string& purpose, const string& policy_store_name) and ~cc_trust_manager()`. The constructor arguments are the enclave type (e.g. - "sev-enclave"), the purpose of the enclave "authentication" or "attestation" and the location of the policy store which stores all your sensitive data between invocations. Most enclaves are of the "authentication" type and they user their certified public keys to authenticate themselves to other certified programs. Some programs can provide confidential computing support to other programs in a trusted boundary and their public keys are certified by the certifier service for attestation. For example, the `application_service` can provide confidential computing support for individual applications within an "encrypted virtual machine."

There is an initialization program that initializes an indicated confidential computing platform. For example, `initialize_sev_enclave_data` initialized AMD-SEV enclaves. The arguments to these functions vary based on the platform but they usually consist of external certificates provided by the platform to establish trust in its properties.

`bool save_store()`: This serializes, encrypts (using the platform `seal` and `unseal` primitieves) and stores the encrypted policy store.

`bool fetch_store()`: This reads and decrypts a policy store for use after restart.

`bool cold_init(const string& public_key_alg, const string& symmetric_key_alg, int policy_cert_size, byte* der_policy_certcert,const string& home_domain_name, const string& home_host,int home_port, const string& service_host, int service_port)`: This generates the applications public, private and symmetric keys. It is used when the program first starts on a new platform. You must call `certify_me` after a `cold_init`. to obtain an admissions certificate for the public key (this is called "certification").

`bool warm_restart()`: This restores the policy store and retrieves all the key data after a restart; is assumes that you have called `certify_me` in an earlier invocation and hence has access to your admissions certificate so you need not and should not call `certify_me` after a `warm restart`.

`bool certify_me()`: This constructs evidence (including an attestation naming your public key) and sends it to the certifier service for evaluation, if successful, it adds the certifier service produced admissions certificate for later use. The ip address of the certifier service and its port, the service address and port and the domain_name were supplied in `cold_init`.

The policy certificate for your application in der encoded form can be retrieved from the class variable `serialized_policy_cert_`.

The remaining interfaces and variables in this class are for internal use.

`class secure_authenticated_channel`: This class opens and manages a mutually authenticated, encrypted and integrity protected channel with other certified programs in your security domain (as identified by the policy key). The calls are:

The constructor `secure_authenticated_channel(string& role)`: The role is either "client" or "server" depending on whether you are opening the channel in an ssl client style mode or server style mode. The corresponding destructor is `~secure_authenticated_channel()`.

`bool load_client_certs_and_key()`: This loads the certs and keys for a client. You need only follow the stylized use in simple example for this.

`bool init_client_ssl(const string& host_name, int port, string& asn1_root_cert, key_message& private_key, const string& private_key_cert)`: This initializes the root policy certificate (which is the "trusted root" for policy establishment), the applications private authentication key (the private key corresponding to the public key in the admissions certificate), and the admissions certificate (signed by the policy key). These are used to establish a secure channel.

`bool init_server_ssl(const string& host_name, int port, string& asn1_root_cert, key_message& private_key, const string& private_key_cert)` does precisely the same initialization for the server side of a channel.

`void server_channel_accept_and_auth(void (*func)(secure_authenticated_channel&))`: This is a stylized server loop to service client requests on a server. See simple example for its use.

`int read(string* out)`, `int read(int size, byte* b)`, `int write(int size, byte* b)`, and `void close()`: These respectively read data into a string from an open channel, read data into a buffer from an open channel, write data into an open channel from a buffer and close a channel.

`bool get_peer_id(string* out)`: This is a very useful function which retrieve the unforgeable peer identity (usually a measurement) for the authenticated peer on the remote end of the channel. You can use this for more granular access decisions to individual resources, for example.

`class policy_store`: This is used to store critical data between invocations of an application and is used by `cc_trust_manager` class to store all its data. You need not use this class directly but can if you need a small key-value store. Most of the calls are self-explanatory but you can consult `certifier_tests` as a reference as well as the examples under `sample_apps`.

In more complex applications, you may want to use functions we employ to implement the functionality in the `certifier_framework.h` functions. We try to maintain inter-release compatibility with functions we think you might wish to use identified in the `certifier_utilities.h` include file. However, you are not required to use these functions. Here are descriptions of most of the ones you might use.

```
bool Seal(const string& enclave_type, const string& enclave_id,
  int in_size, byte* in, int* size_out, byte* out);
bool Unseal(const string& enclave_type, const string& enclave_id,
  int in_size, byte* in, int* size_out, byte* out);
bool Attest(const string& enclave_type,
  int what_to_say_size, byte* what_to_say,
       int* size_out, byte* out);
```

These three provide platform independent access to seal, unseal and attest functionality. For attestation, "what_to_say" is a serialized user_data protobuf.

```
bool protect_blob(const string& enclave_type,key_message& key, int
size_unencrypted_data, byte* unencrypted_data,int*
size_protected_blob, byte* blob);
bool unprotect_blob(const string& enclave_type, int
size_protected_blob, byte* protected_blob,key_message* key, int*
size_of_unencrypted_data, byte* data);
```

Protect_Blob encrypts and interity protects a buffer using the provided key, seals the key and produces a protobuf that Uprotect_blob can recover.

```
bool write_file(const string& file_name, int size, byte* data);
int file_size(const string& file_name);
bool read_file(const string& file_name, int* size, byte* data);
bool read_file_into_string(const string& file_name, string* out);
```

These are simple atomic call file management helpers.

`bool digest_message(const char* alg, const byte* message, int message_len,byte* digest, unsigned int digest_len)`: This produces a cryptographic hash (using the named hash) of the data in the buffer.

```
bool authenticated_encrypt(const char* alg, byte* in, int in_len,
byte *key, byte *iv, byte *out, int* out_size);
bool authenticated_decrypt(const char* alg, byte* in, int in_len,
byte *key, byte *out, int* out_size);
```

These two functions provide authenticated encryption and decryption of buffers. Supported algorithms are "`aes-256-cbc-hmac-sha256`" and "`aes-256-gcm`". The latter uses a 16 byte tag.

`int cipher_block_byte_size(const char* alg_name)`: This returns the size in bytes of the named algorithm.

`int cipher_key_byte_size(const char* alg_name)`: This returns the key size in bytes of the named algorithm.

`int digest_output_byte_size(const char* alg_name)`: This returns the output size in bytes of the named algorithm.

`bool asn1_to_x509(const string& in, X509 *x)`: This converts the DER encoded input (a Google style binary string) into an openssl X509 certificate.

`bool x509_to_asn1(X509 *x, string* out)`: This converts an X509 certificate into its DER encoding.

`bool key_to_RSA(const key_message& k, RSA* r)`: This converts the protobuf encoded key structure into an openssl RSA key (if the key type is an RSA type).

`bool RSA_to_key(const RSA* r, key_message* k)`: This converts an openssl RSA key into a protobuf encoded key.

`void print_protected_blob(protected_blob_message& pb)`: This prints a protected plob.

`int sized_pipe_read(int fd, string* out)`: This reads into a Google style binary output string.

`int sized_pipe_write(int fd, int size, byte* buf)`: This should be clear.

`int sized_ssl_read(SSL* ssl, string* out)`: This reads into a Google style binary output string from an ssly channel.

`int sized_ssl_write(SSL* ssl, int size, byte* buf)`: This should be clear.

`bool get_random(int num_bits, byte* out)`: Gets cryptographic random bits.

`bool time_now(time_point* t)`: Returns a protobuf representation of the currenty date/time now.

`bool time_to_string(time_point& t, string* s)`: Returns a printable time string.

```
bool string_to_time(const string& s, time_point* t);
bool add_interval_to_time_point(time_point& t_in, double hours,
time_point* out)
```

: Adds the indicated time interval to the input date/time to produce the output time.

`int compare_time(time_point& t1, time_point& t2)`: Compares two protobuf encoded times. Returns 1 if t1 is later than t2, 0 if they are the same, -1 if t2 is later than t1.

`void print_time_point(time_point& t)`: Prints a protobuf encoded time.

`void print_key(const key_message& k)`: Prints a protobuf encoded key.

`void print_bytes(int n, byte* buf)`: Prints a byte array.

`bool produce_artifact(key_message& signing_key, string& issuer_name_str, string& issuer_description_str, key_message& subject_key, string& subject_name_str, string& subject_description_str, uint64_t sn, double secs_duration, X509* x509, bool is_root)`: Produces a signed X509 certificate.

`bool verify_artifact(X509& cert, key_message& verify_key, string* issuer_name_str, string* issuer_description_str, key_message* subject_key, string* subject_name_str, string* subject_description_str, uint64_t* sn)`: Verifies a signed X509 certificate.

`bool time_t_to_tm_time(time_t* t, struct tm *tm_time)`: Translate time in time_t format to time in struct tm format.

`bool tm_time_to_time_point(struct tm* tm_time, time_point* tp)`: Translates tim in tm format to time in the certifier time format (time_point).

`bool asn1_time_to_tm_time(const ASN1_TIME* s, struct tm *tm_time)`: Converts time in ASN1_TIME format (from certificates) to time_point format.

`bool get_not_before_from_cert(X509* c, time_point* tp)`: Retrieves not-before time in certificate in time_point format.

`bool get_not_after_from_cert(X509* c, time_point* tp)`: Retrieves not-after time in certificate in time_point format.

`bool add_interval_to_time_point(time_point& t_in, double hours, time_point* out)`: Adds hours to the `t_in` and puts it in `out`.

`void print_time_point(time_point& t)`: Prints time represented in time_point format.

There are many other support files which are not intended for application use and are generally not inter-release compatible. However, from time to time, we may "promote" some of these into `certifier_utilities` if people find them to be "generally useful."

In addition, there is an automatically generated header file for the protobufs and you will find the protobuf definitions in `src/certifier.proto`. We use Google protobufs as our principal serialization format which ensures compatibility while simplifying adding additional information. Protobuf formats are used for communications, key serialization and even rule serialization. An advantage of protobufs is that there it is does not require a general parser and hence parser vulnerabilities can be avoided. Protobufs are also a basic mechanism for capturing data formats that must be understood by different parties. 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.

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.

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` (You can also use the one-step shell script in run_examples.sh). In each example, we compile all the certifier files along with the app using the make file `example_app.mak` rather than linking the certifier library. 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. Incidentally, if you wish to compile the certifier library and link it into the examples, you may do so. You make the certifier library by running `make -f certifier.mak` in the `src` directory.

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 Gramine 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.

Since the instructions were originally written, we've added two new features. The first is a consolidated shell script, in `sample_apps/run_example.sh`, to compile, provision, and run the samples. The shell script can do everything automatically or produce runnable step commands in the style of instructions.txt. This greatly reduces the time to build and run the examples. In addition, there is a new utility called policy_generator that allows you to specify policy in a json like format rather than using shell scripts. As with run_example.sh, the generator can also produce the policy generating shell commands for compatibility.

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, …)
2. app_trust_data->certify_me()
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 another 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. For details, consult `sample_apps/simple_app`.

We have also provided an example illustrating key rotation for keys in `example_key_rotation.cc` and the tests in `support_tests.cc` show how to rotate keys in "protected blobs."

Finally, there is support for getting a single program certified in many security domains. Consult `simple_apps/multidomain_simple_app` for details.

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_manager(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 it when collaborating with other "trusted programs" in its security domain. In the application service, the role is "attestation." Here, 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.
    - 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.

`example_app.cc` uses a helper (`secure_authenticate_channel`) 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) and uses the Admissions Certificate obtained from the Certifier Service.

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 (for example, 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.