Using the Certifier Framework for Confidential Computing

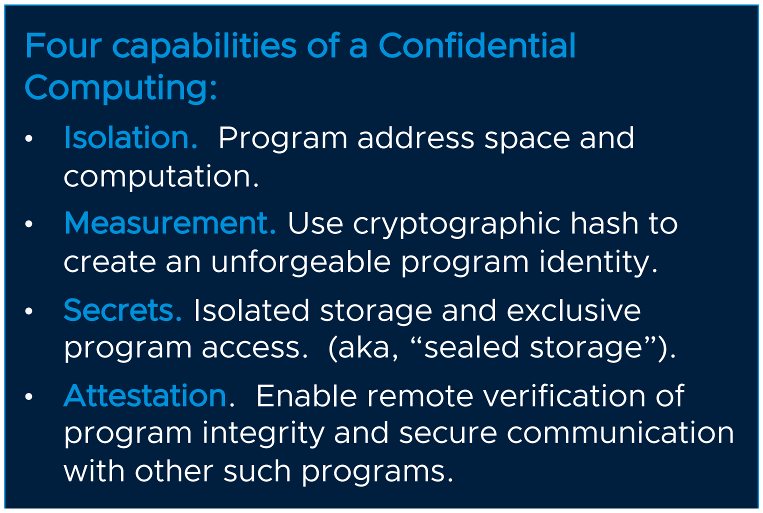
Major Concepts

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).

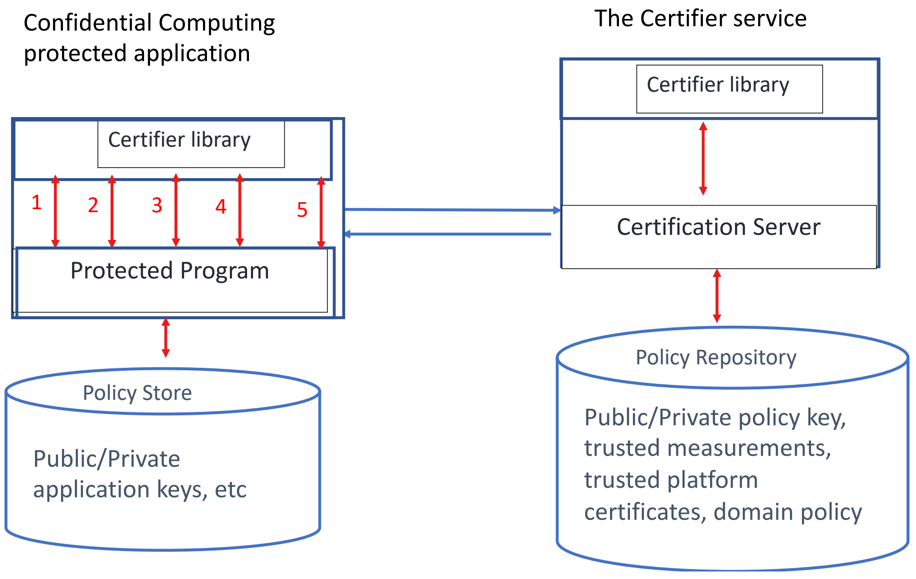

Confidential Computing protected application
The Certifier service
Certifier library
Certifier library
1
2
3
4
5
Protected Program
Certification Server
Policy Store
Public/Private application keys, etc
Policy Repository
Public/Private policy key, trusted measurements, trusted platform certificates, domain policy

Figure 1: Certifier API and Certifier Service

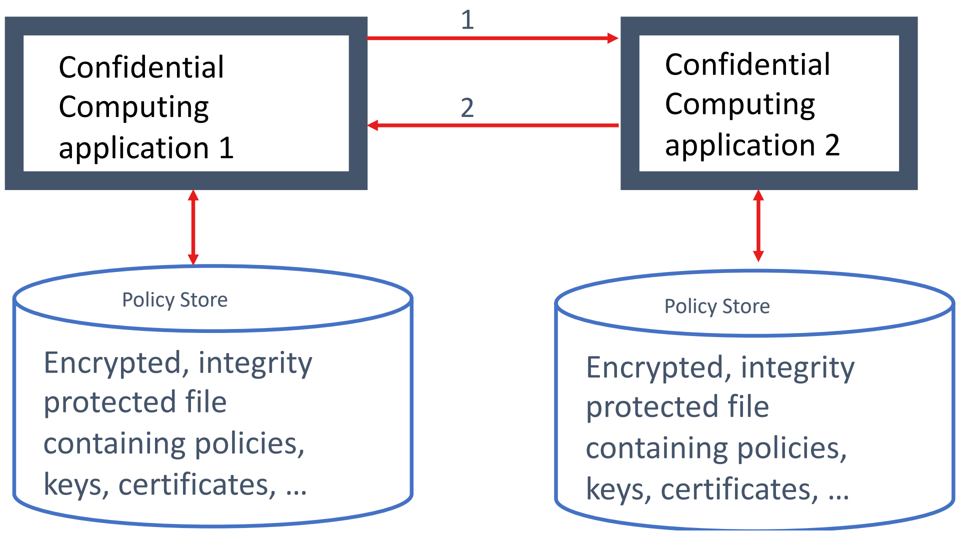

Confidential Computing application 1
1
2
Confidential Computing application 2
Policy Store
Encrypted, integrity protected file containing policies, keys, certificates, …
Policy Store
Encrypted, integrity protected file containing policies, keys, certificates, …

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.

Secondly, in a client, it can run in the following ways:

1. In an SGX enclave under Open Enclaves, Asylo, or Gramine. Here the application enclave is the isolated a measured security principal.
2. In a "certifier client service" (e.g.- a linux daemon running as root) in an encrypted virtual machine under SEV or TDX. Here the entire kernel and initramfs is the isolated and measured security principle. The "certifier client service" can provide service for OS-wide actions.
3. In an application (process) in an encrypted virtual machine under SEV or TDX. Here, the process is the isolated and measured principal. The OS-wide "certifier client service" provides Isolation, Measurement, Sealing, Unsealing and Attestation services for the application using keys within the "certifier client service;" each such protected program is a independent Confidential Computing program isolated from the others by OS level process isolation. All such programs are descendants (e.g-children) of the "certifier client service". In this case, attestation consists of two parts: the OS attestation of the certifier client service performed by the hardware and the process level attestation performed by the certifier client service.

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

Writing Applications

The certifier API makes converting a well written application into a Confidential Computing enabled application easy.

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 "sample_app." In the sample_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 develops all the policy and applications that use it (`example_app.cc`, in this case) need know nothing about the policy details.

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 and a self-signed cert for the key. The private portion of the policy key is used only by the
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_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. Thus, 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 and authentication key, called an enclave key, which allows it to authenticate it to other programs in the security domain and puts the private and public portions of the enclave key in the policy store. This is done in `cold_init_trust_info()` in `example_app.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.
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 at the Certifier Service and sends back a `trust_request_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 `example_app.cc`.

5. The Confidential Computing application extracts the admission certificate 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 obtains its policy store (which is decrypted, and integrity checked by the certifier) and retrieved 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 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 no changes.
8. During execution, a Confidential Computing application may transmit secrets or data with another 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_app and the instructions there provide a complete step by step guide to writing Confidential Computing Applications and deploying them.

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 with other Confidential Computing programs to rely on (trust) their security properties must first submit evidence (including attestations) to establish 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 sample application provided with this repository in `sample_app/instructions.txt`.

Proofs from the Certifier (or 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 and these statements are called "`vse-clauses`." You can also use other claim formats or substitute another policy evaluation engine, like Datalog or OPA as the indicated in the code.

The internal 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; it is produced by the policy tools in the utilities directory. Consult `sample_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 are format agnostic. Any tokens or formats you use in an application or service work the same way. No need for token translation a change in application authorization logic.
2. Provisioning of keys and data require almost no change. 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 in the same was except through a secure channel.
3. The certifier relies on no root key store, so application actions are entirely controlled by signed policy.
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. The 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. There is no dependency on service providers. Neither improper configuration within a service provider (or on your own machines!), 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 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.

Some Applications

Here are some applications:

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_app` which gives a complete picture of all aspects of using the certifier and certifier service.

Appendix --- API

Below is an API list; however, you will likely use only a few of these in any application, namely, the Confidential Computing Primitives and the Policy Store.

Crypto and file support

```
bool write_file(string file_name, int size, byte* data);
int file_size(string filename);
bool read_file(string file_name, int* size, byte* data);
bool encrypt(byte* in, int in_len, byte *key, byte *iv, byte *out, int*
out_size);
bool decrypt(byte *in, int in_len, byte *key, byte *iv, byte *out, int*
size_out);
bool digest_message(const byte* message, int message_len, byte* digest,
unsigned int digest_len);
bool authenticated_encrypt(byte* in, int in_len, byte *key, byte *iv, byte
*out, int* out_size);
bool authenticated_decrypt(byte* in, int in_len, byte *key, byte *out, int*
out_size);
bool asn1_to_x509(string& in, X509 *x);
bool x509_to_asn1(X509 *x, string* out);
bool make_root_key_with_cert(string& type, string& name, string& issuer_name,
key_message* k);
bool make_certifier_rsa_key(int n,  key_message* k);
bool rsa_public_encrypt(RSA* key, byte* data, int data_len, byte *encrypted,
int* size_out);
bool rsa_private_decrypt(RSA* key, byte* enc_data, int data_len, byte*
decrypted, int* size_out);
bool rsa_sha256_sign(RSA*key, int size, byte* msg, int* size_out, byte* out);
bool rsa_sha256_verify(RSA*key, int size, byte* msg, int size_sig, byte*
sig);
bool generate_new_rsa_key(int num_bits, RSA* r);
bool key_to_RSA(const key_message& k, RSA* r);
bool RSA_to_key(RSA* r, key_message* k);
bool private_key_to_public_key(const key_message& in, key_message* out);
bool get_random(int num_bits, byte* out);
void print_bytes(int n, byte* buf);
void print_key_descriptor(const key_message& k);
void print_entity_descriptor(const entity_message& e);
void print_vse_clause(const vse_clause c);
void print_claim(const claim_message& claim);
void print_signed_claim(const signed_claim_message& signed_claim);
void print_storage_info(const storage_info_message& smi);
void print_trusted_service_message(const trusted_service_message& tsm);
void print_attestation(attestation& at);
void print_protected_blob(protected_blob_message& pb);
bool vse_attestation(string& descript, string& enclave_type, string&
enclave_id,vse_clause& cl, string* serialized_attestation);
bool make_signed_claim(const claim_message& claim, const key_message& key
signed_claim_message* out);
bool verify_signed_claim(const signed_claim_message& claim, const
key_message& key);
```

```
bool verify_signed_attestation(int serialized_size, byte* serialized, int
sig_size, byte* sig, const key_message& key);
bool time_now(time_point* t);
bool time_to_string(time_point& t, string* s);
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);
int compare_time(time_point& t1, time_point& t2);
void print_time_point(time_point& t);
void print_entity(const entity_message& em);
void print_key(const key_message& k);
void print_rsa_key(const rsa_message& rsa);
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 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);
```

Policy store

```
class policy_store {
public:
  policy_store();
  policy_store(int max_trusted_services, int max_trusted_signed_claims, int
max_storage_infos, int max_claims, int max_keys);
  ~policy_store();

  bool replace_policy_key(key_message& k);
  const key_message* get_policy_key();

  int get_num_trusted_services();
  const trusted_service_message* get_trusted_service_info_by_index(int n);
  int get_trusted_service_index_by_tag(string tag);
  bool add_trusted_service(trusted_service_message& to_add);
  void delete_trusted_service_by_index(int n);

  int get_num_storage_info();
  const storage_info_message* get_storage_info_by_index(int n);
  bool add_storage_info(storage_info_message& to_add);
  int get_storage_info_index_by_tag(string& tag);
  void delete_storage_info_by_index(int n);

  int get_num_claims();
  const claim_message* get_claim_by_index(int n);
  bool add_claim(string& tag, const claim_message& to_add);
  int get_claim_index_by_tag(string& tag);
  void delete_claim_by_index(int n);

  int get_num_signed_claims();
  const signed_claim_message* get_signed_claim_by_index(int n);
  int get_signed_claim_index_by_tag(string& tag);
  bool add_signed_claim(string& tag, const signed_claim_message, to_add);
  void delete_signed_claim_by_index(int n);

  bool add_authentication_key(string& tag, const key_message& k);
  const key_message* get_authentication_key_by_tag(string& tag);
  const key_message* get_authentication_key_by_index(int index);
  int get_authentication_key_index_by_tag(string& tag);
  void delete_authentication_key_by_index(int index);
  bool Serialize(string* out);
  bool Deserialize(string& in);
  void clear_policy_store();
};
void print_store(policy_store& ps);
```

Clauses, claims and signed claims

```
bool same_key(const key_message& k1, const key_message& k2);
bool same_measurement(string& m1, string& m2);
bool same_entity(const entity_message& e1, const entity_message& e2);
bool same_vse_claim(const vse_clause& c1, const vse_clause& c2);
bool make_key_entity(const key_message& key, entity_message* ent);
bool make_measurement_entity(string& measurement, entity_message* ent);
bool make_unary_vse_clause(const entity_message& subject, string& verb,
vse_clause* out);
bool make_simple_vse_clause(const entity_message& subject, string& verb,const
entity_message& object, vse_clause* out);
bool make_indirect_vse_clause(const entity_message& subject, string& verb,
const vse_clause& in, vse_clause* out);
bool make_claim(int size, byte* serialized_claim, string& format, string&
descriptor, string& not_before, string& not_after, claim_message* out);
```

Policy statement and verification

```
bool init_certifier_rules(certifier_rules& rules);
bool init_axiom(key_message& pk, proved_statements* _proved);
bool init_proved_statements(key_message& pk, signed_assertions& assertions,
proved_statements* shown);
bool convert_attestation_to_vse_clauseconst (claim_message& sa, vse_clause*
cl);
bool verify_signed_assertion(const key_message& key, const
signed_claim_message& sc, vse_clause* cl);
bool verify_external_proof_step(proof_step& step);
bool verify_internal_proof_step(const vse_clause s1, const vse_clause s2,
const vse_clause conclude, int rule_to_apply);
bool statement_already_proved(const vse_clause& cl, proved_statements*
are_proved);
bool verify_proof(key_message& policy_pk, vse_clause& to_prove,
signed_assertions& signed_statements, int num_steps, local_proof_step
*the_proof, proved_statements* are_proved);
void print_trust_response_message(trust_response_message& m);
      void print_trust_request_message(trust_request_message& m);
```

Confidential computing primitives

```
bool Getmeasurement(string& enclave_type, string& enclave_id,int* size_out,
byte* out);
bool Seal(string& enclave_type, string& enclave_id, int in_size, byte* in,
int* size_out, byte* out);
bool Unseal(string& enclave_type, string& enclave_id, int in_size, byte* in,
int* size_out, byte* out);
bool Attest(string& enclave_type, int what_to_say_size, byte* what_to_say,
int* size_out, byte* out);
bool Protect_Blob(string& enclave_type, key_message& key, int
size_unencrypted_data, byte* unencrypted_data, int* size_protected_blob,
byte* blob);
bool Unprotect_Blob(string& enclave_type, int size_protected_blob, byte*
protected_blob, key_message* key, int* size_of_unencrypted_data, byte* data);
```