Universally Composable Cryptographic Library

Recall the Dolev-Yao model

- The messages were terms (trees); elements of a free algebra.
  - Certain values were represented as atomic messages
    - keys, nonces, user's secrets, (random coins)
  - There were constructors that made messages from messages
    - pairing (tupling), encryption, signatures, (MACs, etc.)
- There were certain rules on how the messages could be decomposed, given in terms of the structure of the messages.
  - The adversary was also bound by these rules.
- Secrecy of a message meant that the adversary could not obtain the term corresponding to it.

$\mathcal{TH}_n$ — ideal UC cryptolib for n parties

- Connects to n users and the adversary.
- Main part of the state — a database of terms.
- For each term and each user/adversary:
  - The database records whether this term is known to this user or not.
- If the term is known to the user/adversary, he has a handle for it.
- The handles carry no information about the structure of terms.
  - But for each term and each user/adversary, there is only one handle.
- The users and adversary can create new terms and move downwards in the forest of terms.
- Sending a message to a different user requires translation of handles.

Message manipulation commands

- Store and retrieve payloads.
  - Storing the same payload twice creates just a single entry in the database.
- Construct tuples. Read components of tuples.
  - Constructing the same tuple twice creates just a single entry in the database.
- Generate nonces.
- Public-key encryption: generate keypairs, encrypt, decrypt.
- Signatures: generate keypairs, sign, verify, get message.
- Symmetric encryption: generate keys, encrypt, decrypt.
- MACs: generate keys, tag, verify, get message.
  - Models randomized tagging algorithm.
- (Compare messages) — just compare handles.
- Get the type of a message.

Sending messages

- Messages reside inside $\mathcal{TH}_n$. Accessed through handles.
- The only operation giving a non-handle is retrieval of payloads.
- Transmission of messages has to be handled by $\mathcal{TH}_n$ as well.
- Messages can be sent over secure or insecure channels.
  - In the original formulation, authentic channels existed, too.
- The adversary can impersonate anyone else on insecure channels.
- The adversary schedules the secure channels.
- Secret keys of asymmetric primitives may not be sent.
  - They may only be used for signing or decryption. They cannot be included in messages.

NonDY — message lengths

- Each term in the database has a well-defined length.
- The formula for computing the length of a term from the lengths of its subterms may depend on the security parameter.
- The machine $\mathcal{TH}_n$ only agrees to do polynomial amount of work for each user.
- Each party can query $\mathcal{TH}_n$ for the length of any term that it has the handle for.

NonDY — identities of keys

- Given a signature or a public-key ciphertext, it is possible to get the public key from it.
- Given a MAC or a symmetric ciphertext, the adversary is able to learn the identity of the key from it:
    - The commands gen_symenc_key and gen_mac_key actually create two nodes — the key and its "identity".
        - Only the handle to the key is returned.
        - But the adversary is able to get the handle to the identity as well.
        - Given a ciphertext or a MAC, the adversary can ask for the identity of the used key.

NonDY — abilities of the adversary

The adversary can additionally generate the following nodes:

- Garbage (of special type "garbage").
- Invalid asymmetric ciphertext of given length ℓ.
  - Points to the key, but not to any plaintext.
  - Attempt to decrypt results in error.
- Transformed signatures
  - Given a signature S of text T with the key K, generates a new node S' that is also a signature of text T with the key K.
- Transformed MACs
- A MAC with no key.
  - A new MAC-node M is created, that points to the given text T, but does not point to any tagging keys.
- An empty symmetric ciphertext of given length ℓ.
  - Neither the key nor the plaintext have to be fixed.

NonDY — abilities of the adversary

The adversary can change the already created nodes as follows:

- Given a MAC M, the adversary can add a new key, under which this MAC verifies.
  - The adversary must know that key.
  - Hence, in general, a MAC-node M in the database of $\mathcal{TH}_n$ points to a message m and to zero or more tagging keys.
- Given a symmetric ciphertext, the adversary can add a new pair of (key,plaintext), such that this ciphertext decrypts to the given plaintext under the given key.
  - The adversary must know the key and the plaintext.
  - The key must not yet be a valid key of this ciphertext.
  - In general, a symmetric encryption node SE contains a list of pairs, each of them pointing to a symmetric key and a message.

When the adversary asks for the identity of the key of some symmetric ciphertext or MAC, he gets a list of identities.

The real system

- One machine M_i for each of the parties $i \in \{1, \ldots, n\}$.
- Connected to the i-th user, the adversary and also to all other machines (for implementing secure channels).
- Internally, the machine M_i contains a list of pairs (handle, bit-string) mapping handles to actual messages.
  - Each message must contain its type.
- The library works pretty much as you imagine.
- Potential pitfall — no bit-string may have several different handles.
- Use random bit-strings as key identities. If the key is used in a message, pair it with its identity. Append each MAC or symmetric encryption with the identity of the key.
- Add the public key to all public-key ciphertexts and signatures.

The simulator

- The job of the simulator is to translate between the terms in the ideal system and the bit-strings in the real system.
- During its work it builds up a database of triples $(hnd, w, args)$ where w is a bit-string and hnd is the handle for the ideal adversary.
- $args$ contains additional information, for example the signing keys.
- This database serves as the dictionary.

Translating ideal $\rightarrow$ real

The simulator has received a new handle from $\mathcal{TH}_n$ and has to produce a bit-string corresponding to it.

- Parse the ideal message as much as possible. Enter new payloads, generate new nonces, keys, ciphertexts, signatures, MACs as necessary.
- Whenever we see a handle (hnd) for a new verification key, generate a new signing keypair (sk, vk) and store (hnd, vk, sk).
  - Use sk to generate signatures that are verifiable with hnd.
- Same for public encryption keys and key identities.
  - For identities of keys — we may later get the handle to the key itself, too.
- If we see the handle to a ciphertext, such that we do not have the handle to the decryption key, then we encrypt a random bit-string of correct length.

Translating real → ideal

Simulator received a bit-string w and has to find a handle.

- Parse the bit-string as much as possible. Enter the new values in the databases of $\mathcal{TH}_n$ and simulator.
- When the bit-string w an unseen verification key, then ask $\mathcal{TH}_n$ to create a new signing keypair $(hnd_{\mathrm{sk}}, hnd_{\mathrm{vk}})$ Add (hnd_{vk}, w, sk) to simulator's database.
- Same for public encryption keys.
- Translating a signature:
    - If the simulator has the handle to the signing key, then ask $\mathcal{TH}_n$ to create a new signature.
    - Otherwise, if the simulator has the handle to a different signature of the same message with the same key, ask $\mathcal{TH}_n$ to transform this signature.
    - Otherwise give up.

Translating real $\rightarrow$ ideal

- Translating a public-key ciphertext:
  - If the simulator does not know the secret key, then ask $\mathcal{TH}_n$ to create an invalid ciphertext.
  - If the secret key is known, but the plaintext does not make sense, then also ask $\mathcal{TH}_n$ to create an invalid ciphertext.
  - Otherwise ask $\mathcal{TH}_n$ to create a real ciphertext.
- Translating a tagging key: for all MACs received so far, consider whether this key w successfully verifies them. If yes, then ask $\mathcal{TH}_n$ to add hnd_{sk} to this tag as a verification key.
- Translating a MAC:
  - If we do not know a the secret key yet and the message is new (for this key identity), then as $\mathcal{TH}_n$ to add a MAC with no verification keys.
- Translating symmetric keys and ciphertexts: similar.

The commitment problem

Simulation of symmetric encryption does not always work.
Simulator fails if a user does the following:

$$
\begin{aligned}
&\mathtt{k} \leftarrow \mathit{new\_symmetric\_key}\\
&\mathtt{x} \leftarrow \mathit{payload}(M)\\
&\mathtt{y} \leftarrow \mathit{sym\_encrypt}(\mathtt{k}, \mathtt{x})\\
&\textbf{send } \mathtt{y}\\
&\textbf{send } \mathtt{x}\\
&\textbf{send } \mathtt{k}
\end{aligned}
$$

- Translate `y`: generate $k \leftarrow \mathcal{K}_s()$, $z \leftarrow \mathit{rand\_string}$, $y \leftarrow \mathcal{E}_s(k, z)$.
- Translate `x`: $x \leftarrow M$ (given `x`, simulator can ask for it).
- Translate `k`: ???
  - Translation k must satisfy $x = \mathcal{D}_s(k, y)$.

Restricting the honest user

The simulatability proof goes through if we demand that the honest user

- never causes a key to leak that it has already used;
  - leak in the sense of Dolev-Yao
- avoids encryption cycles.
  - There are several ways to formalize this.
  - Original paper — let $sk_1, sk_2, \ldots$ be all symmetric keys in the order they are first used for encryption. We demand that sk_i is only encrypted by keys sk_j where $j < i$.
  - A later formulation — the command gen_symenc_key contains a parameter i — the "order" of the key. A key of order i is only allowed to encrypt keys of lower order.
- This must be guaranteed by the honest user alone.

On proof of real $\approx$ (ideal$\|$simulator)

- Encapsulate asymmetric encryption and signatures into separate machines $\mathcal{E}nc^n$ and $\mathcal{S}ig^n$. Replace them with their ideal counterparts.
- Do the same for the symmetric encryption.
  - Can only do one key at a time.
  - There must be no encryption cycles.
- Construct the probabilistic bisimulation with error sets. Errors correspond to
  - Collisions in real nonces, keys, etc.
  - The adversary guessing the nonces, keys, etc.
  - The adversary forging a MAC.

Secrecy properties

- Let a structure $\mathcal{S}$ implement a protocol, using the UC cryptolib for cryptographic operations and networking.
- Let H be a user of $\mathcal{S}$.
  - H gives payloads to $\mathcal{S}$; $\mathcal{S}$ transports the payloads between different parties.
- key secrecy: Ideal-system A does not learn the handles of the newly generated keys we're interested in. The view of real-system A is independent from the values of actual keys.
- payload secrecy — The view of $H\|A$ does not distinguishably change, if the following change to the semantics is made:
  - Pick a random length-preserving permutation $\pi : \{0,1\}^* \to \{0,1\}^*$.
  - When H sends M to $\mathcal{S}$, $\mathcal{S}$ receives $\pi(M)$.
  - When $\mathcal{S}$ sends M' back to H, H receives $\pi^{-1}(M')$.

Payload secrecy, symbolically

Theorem. $\mathcal{S}\|\mathcal{TH}_n$ preserve the secrecy of payloads if

- $\mathcal{S}$ passes a payload M down to $\mathcal{TH}$ only as a payload;
- the adversary will not obtain the handle for M;
- M does not affect the control flow of the programs of $\mathcal{S}$.

Very similar to secrecy in the formal model.

Payload secrecy and key secrecy are preserved under simulation.