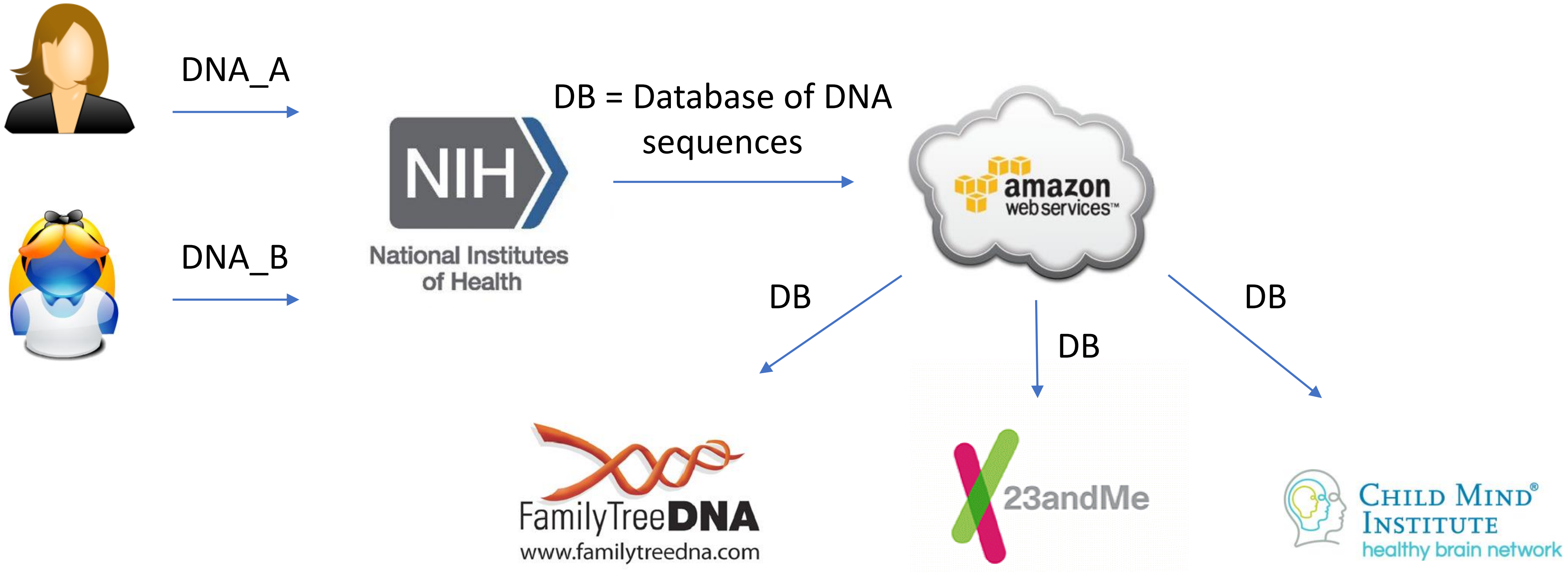
Iron: Functional encryption using Intel SGX

Sergey Gorbunov

University of Waterloo

Joint work with Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh.

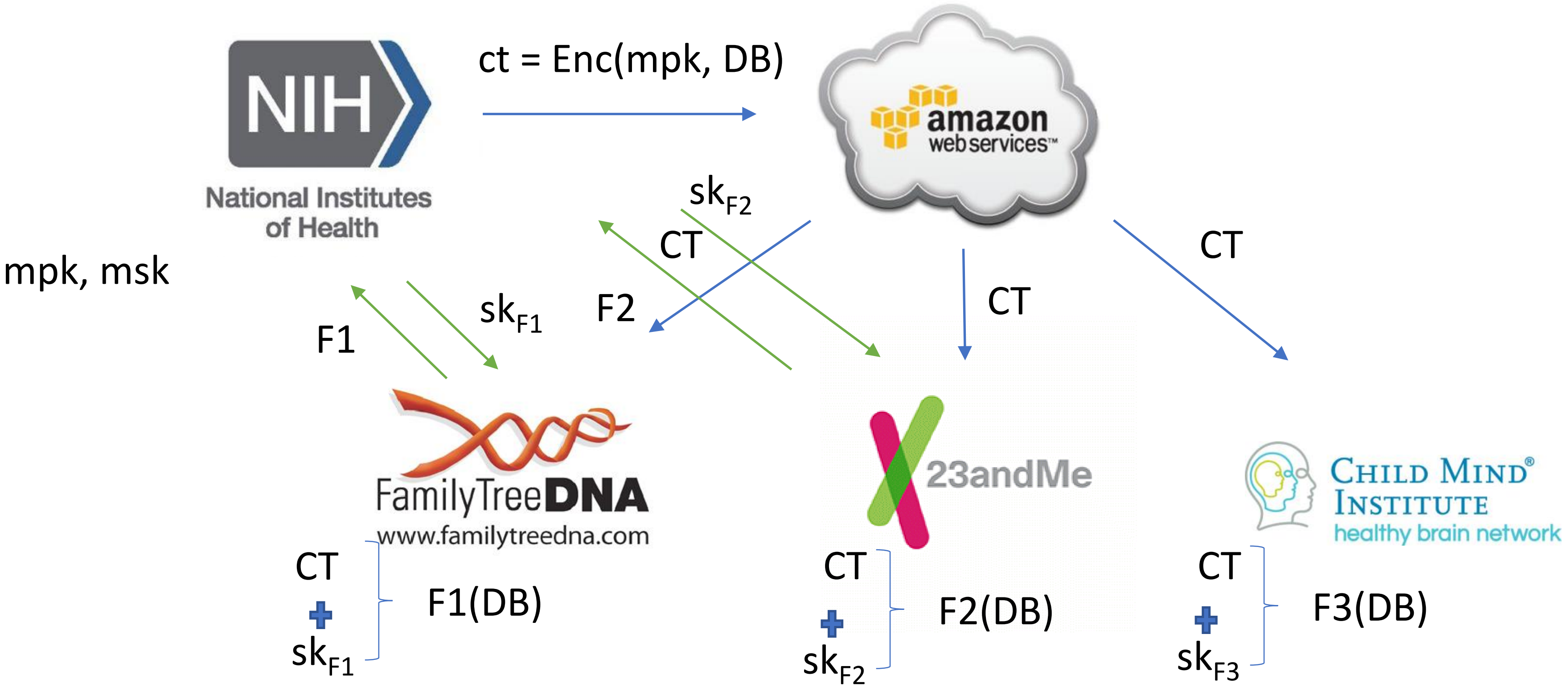
Motivation



Challenges:

1. Ensure privacy of users' DNA sequences in the DB.
2. Selectively enable services (i.e. computations) over private data in DB

FE to the Rescue



FE Definition

[Boneh, Sahai, Waters 11]

- (mpk, msk) ← Setup(1^n) *Authority (NIH)*
- ct ← Enc(mpk, X) *Data Owner (may not be NIH)*
- sk_F ← Keygen(msk, F) *Authority*
- F(X) ← Dec(sk_F, ct) *Service / Data User*

FE Security - Informal

- Simulation (SIM):

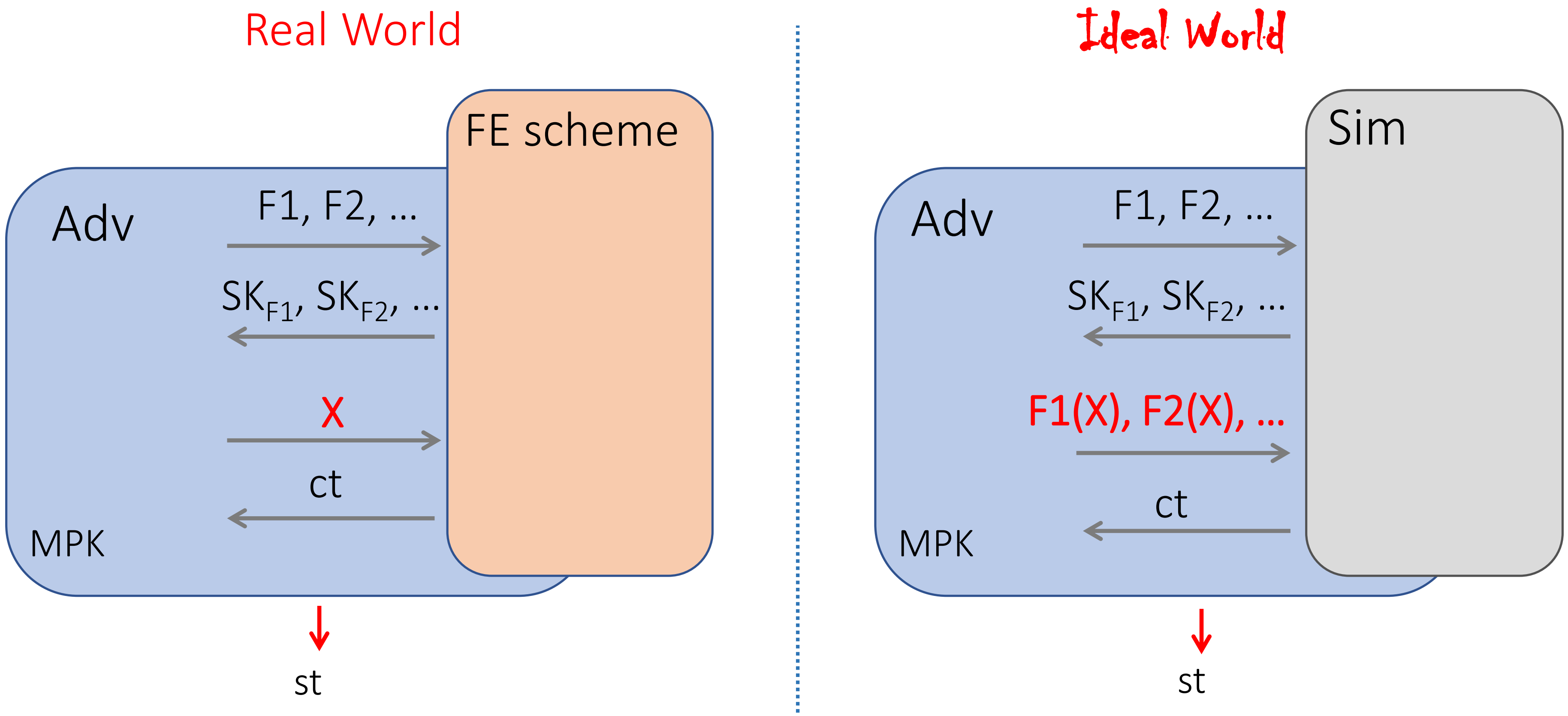
Adversary given (sk_{F1}, sk_{F2}, …, sk_{Fq}) and Enc(mpk, X), learns only F1(X), F2(X), …, Fq(X)

- Indistinguishability (IND):

Adversary given access to (sk_{F1}, sk_{F2}, …, sk_{Fq}), cannot distinguish between Enc(mpk, X_0) and Enc(mpk, X_1) where $F_i(X_0) = F_i(X_1)$ for all i.

FE Security – semi-formal

[BSW11,O'N10]



$$(X, st) \quad {}_{c}{\approx} \quad (X, st)$$

Previous Results

- FE for Boolean formulas/inner products [GPSW06, LOSTW10, AFV11, ABDP15, BJK15, ALS16, KLM+16, BCFG17, …]
  - ✓ Various standard assumptions: LWE, pairings, etc.
  - ✓ Somewhat efficient
- General functions/circuits [GGHRSW14, ABSV15, Wat16, BKS16, BNPW16, …]
  - x Non-standard assumptions (multi-linear maps, obfuscation)
  - x Very inefficient [ACLL'15]

Can we build an *efficient, provably-secure* FE scheme for arbitrary functions from a *plausible* assumption?

Our Results

Thm: We present efficient, provably-secure FE for arbitrary functions assuming existence of secure hardware (Intel SGX) modules.

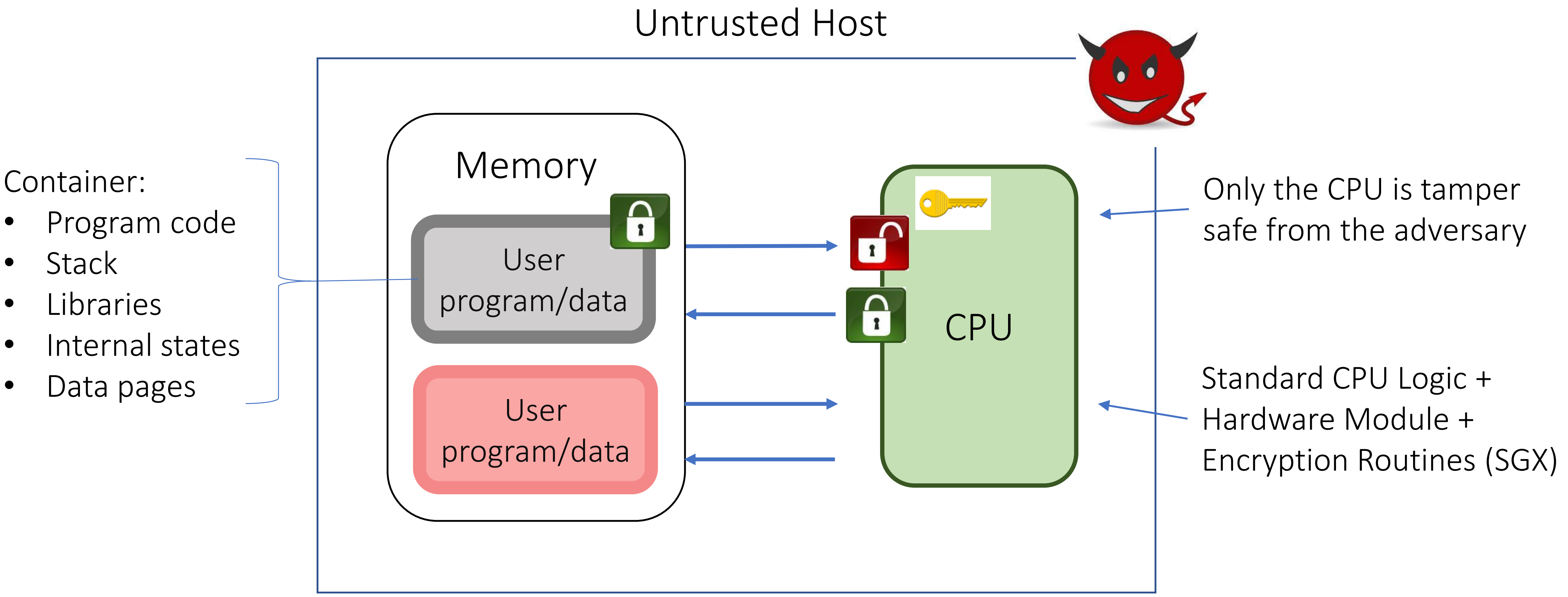
- ✓ We model and argue the security under strong simulation notion.
- ✓ No restriction on the complexity of functions: need to be written in C/C++.
- ✓ We demonstrate practical efficiency with a prototype implementation and benchmark against known crypto FE constructions.

Outline

- ✓ Motivation and our results
- Background on secure hardware (Intel SGX)
- Construction overview
- Proof insights
- Implementation details and performance

Intel SGX Overview

Goal: provide secure execution environment on an untrusted remote host, assuming only security of a processor enabled with a set of encryption routines (Intel SGX).



(steady state, post-setup)

Intel SGX Overview

✓ Encrypted user-level memory container
  - User-level = cannot do syscalls, IO, network communication, etc.

✓ Physically encrypted pages of program code and data in memory

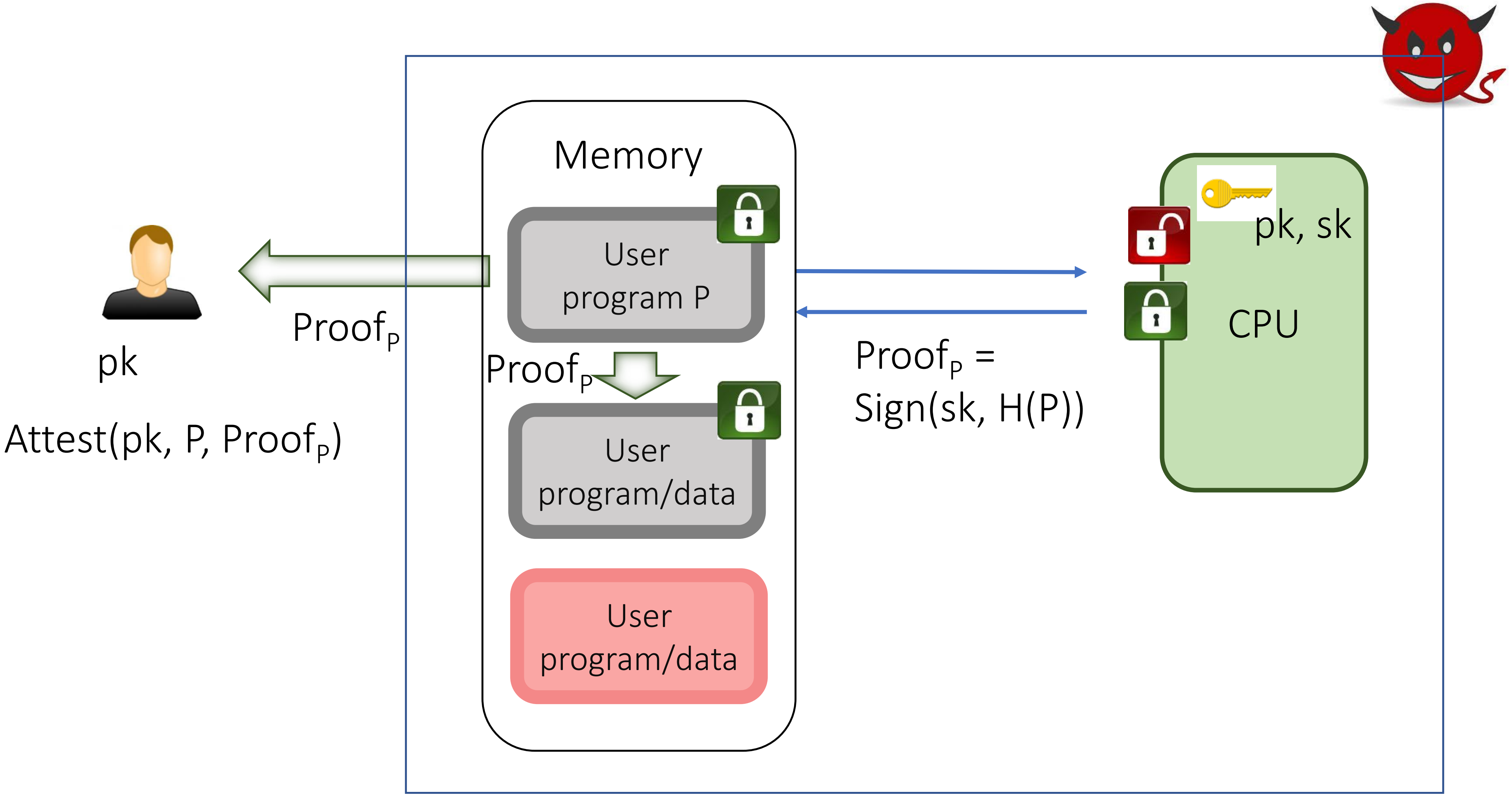
✓ Key is protected on the CPU and cannot be extracted, encrypts/decrypts container pages before execution

Intel SGX Overview

Property 1: Attestation

- A party can verify that it is communicating with a program running in the encrypted container on a platform associated with a key pair (pk, sk)
- Verification wrt a public “measurement” of the program (hash)
- **Local attestation**: two containers running on the same node can attest each other
- **Remote attestation**: a remote user can attest that a specific program is running inside a secure container

Intel SGX Overview

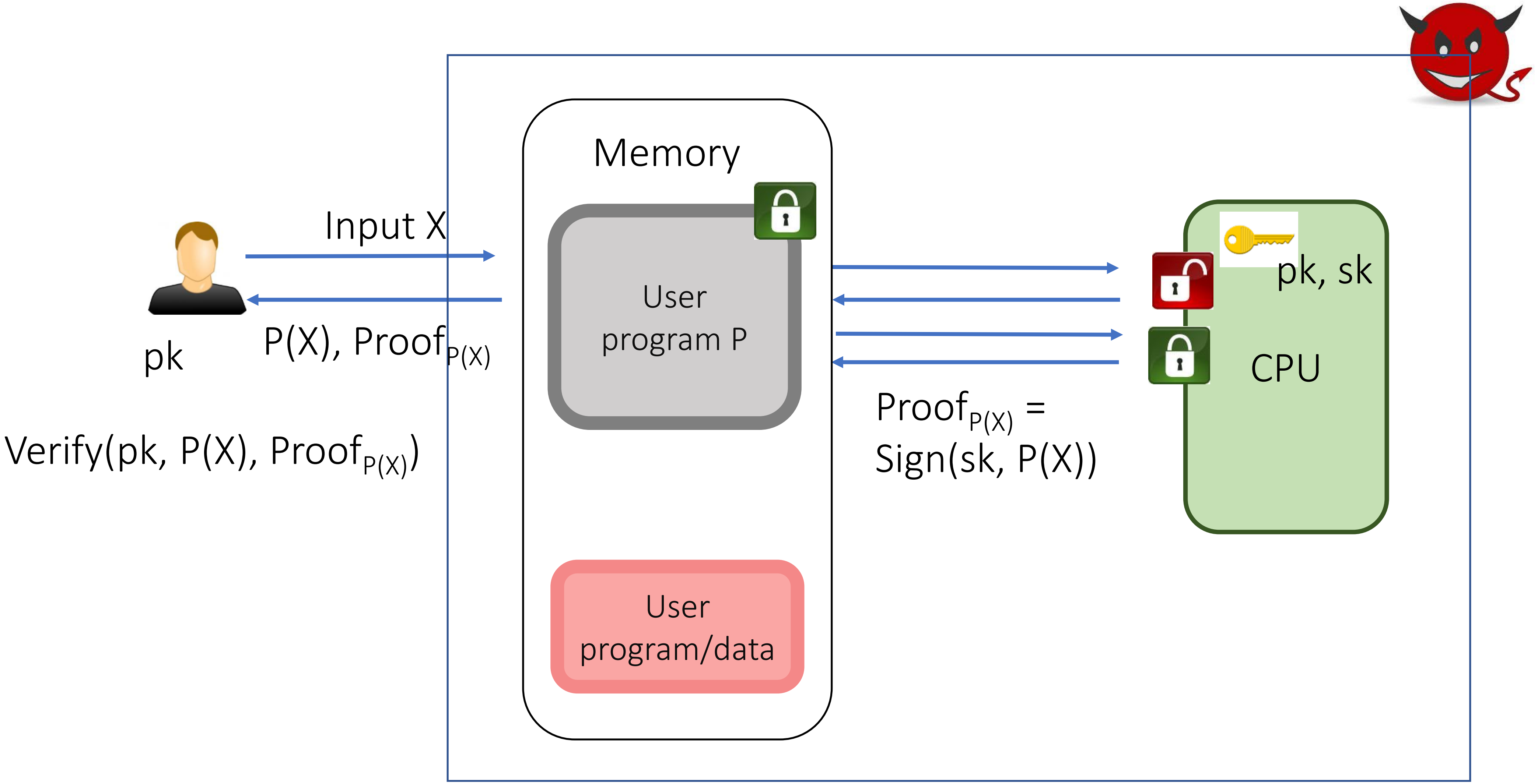


Intel SGX Overview

Property 2: Isolated execution

- *Confidentiality*: “black-box” execution of a program
  - ✓ Internal state of the program is hidden from adversary
- *Integrity*:
  - ✓ Adversary cannot change execution state/data/program,
  - ✓ Cannot modify the output of the program on a given input

Intel SGX Overview

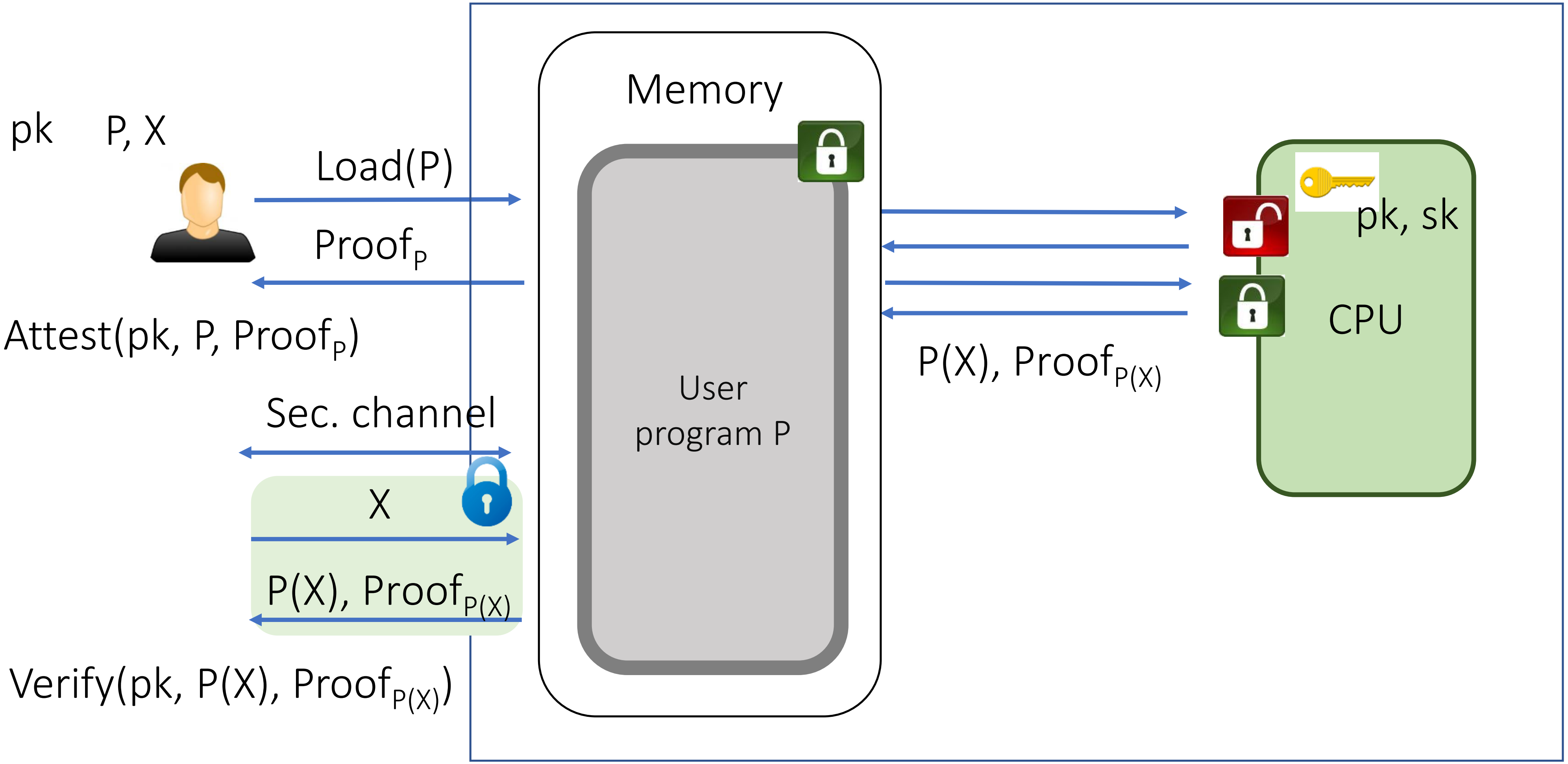


SGX Formal Algorithms

- $\text{Setup}(1^n) \rightarrow (\text{sk}, \text{pk})$
- $\text{Load}_{\text{sk}}(P) \rightarrow \text{Proof}_P$
- $\text{Attest}(\text{pk}, P, \text{Proof}_P) \rightarrow 0/1$
- $\text{Run}_{\text{sk}}(X) \rightarrow (P(X), \text{Proof}_{P(X)})$
- $\text{Verify}(\text{pk}, P(X), \text{Proof}_{P(X)}) \rightarrow 0/1$

SGX Initialization and Runtime

Goal: secure verifiable computation outsourcing of a program P on input X.



SGX – The Good

- Shielded execution of unmodified Windows apps [BPH14]
- Secure MapReduce computations [SCF+15, DSC+15, OCF+15]
- Secure Linux containers [ATG+16, STT+17]
- An authenticated data feed for smart contracts [ZCC+16]
- Secure distributed data analytics (Spark SQL) [ZDB+17]
- Other CPU manufacturers have their own version of SGX (AMD SEV)
- Easy to use, develop, integrate, etc.

Becoming a building block for many secure applications!

SGX – The Ugly

- Programs running inside encrypted containers are subject to side-channel attacks:
  - Page-fault attacks [XCP15]
  - Synchronization bugs [WKPK16]
  - Branch shadowing [SLK+17]
  - Cache attacks [BMD+17, SWG+17]
- Lots of academic work providing stronger security guarantees and mitigating SGX side-channels [CLD16, SLKP16, LSG+16, WKPK16, SLK+17, SGF17].

SGX – The Ugly Cont.

- Intel is trusted for the HW implementation
- Cannot change the working function inside the encrypted container after it is loaded/attested
- Small working memory (~90MB)
- No system calls/IO/network communication

System vs Model vs Proof

IPSec

BLOCKCHAIN

SSL
TLS

Signal Private Messenger

Disk encryption

Tor

Outline

- ✓ Motivation and our results
- ✓ Background on secure hardware (Intel SGX)
- Construction overview
- Proof insights
- Implementation details and performance

Our Construction

(simplified)

Building blocks:

- SGX (on data user node)
- public-key encryption (p.setup, p.enc, p.dec)
- signature scheme (s.setup, s.sign, s.verify)

Our Construction

(simplified)

Enc(mpk, X) → ct

1) $p.enc(pk_p, X) \rightarrow ct$

Setup(1^k) → (mpk, msk)

1) $s.setup(1^k) \rightarrow (vk_s, sk_s)$
2) $p.setup(1^k) \rightarrow (pk_p, sk_p)$
3) $mpk = (pk_p, vk_s)$, $msk = (sk_p, sk_s)$

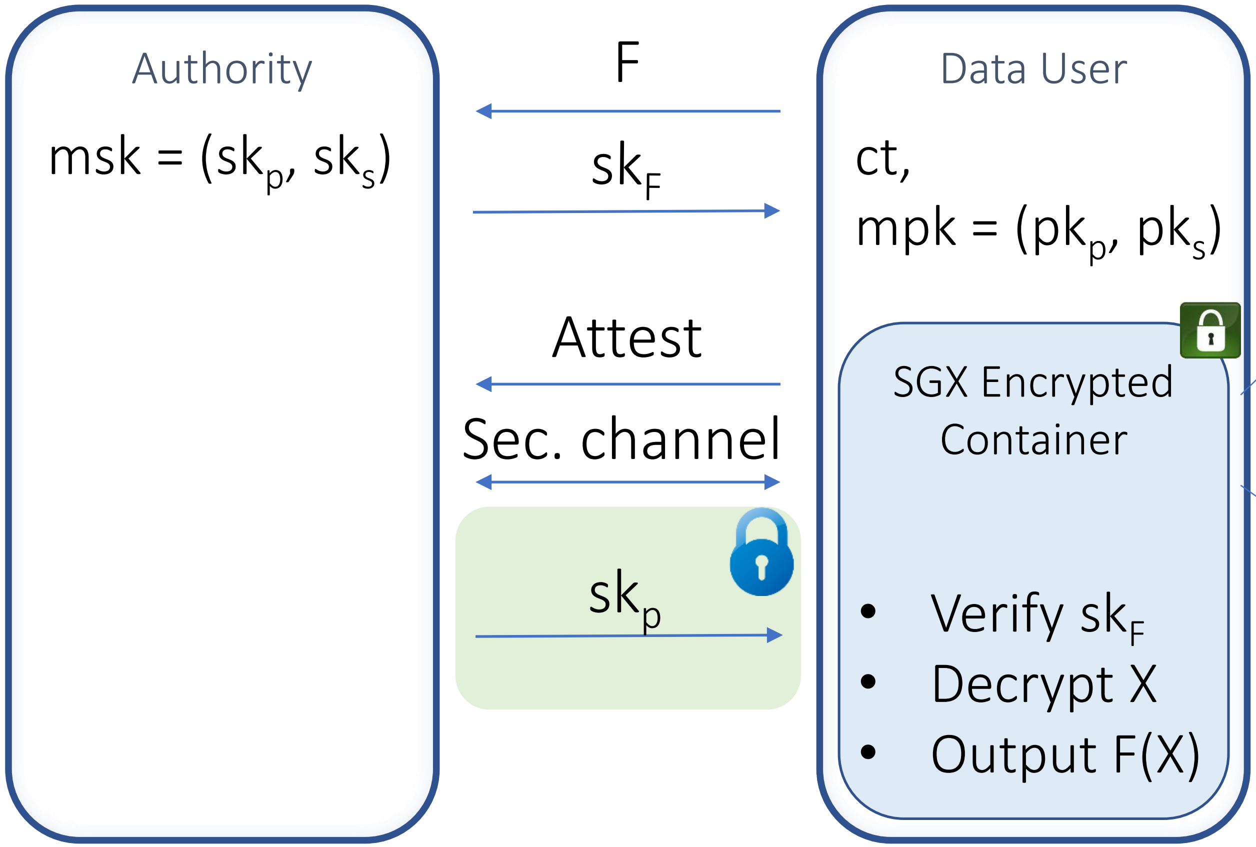
Keygen(msk, F) → sk_F

1) $s.sign(sk_s, F) \rightarrow sk_F$

Dec(sk_F, ct) → F(X)

(next slide)

$Dec(sk_F, ct) \rightarrow F(X)$:



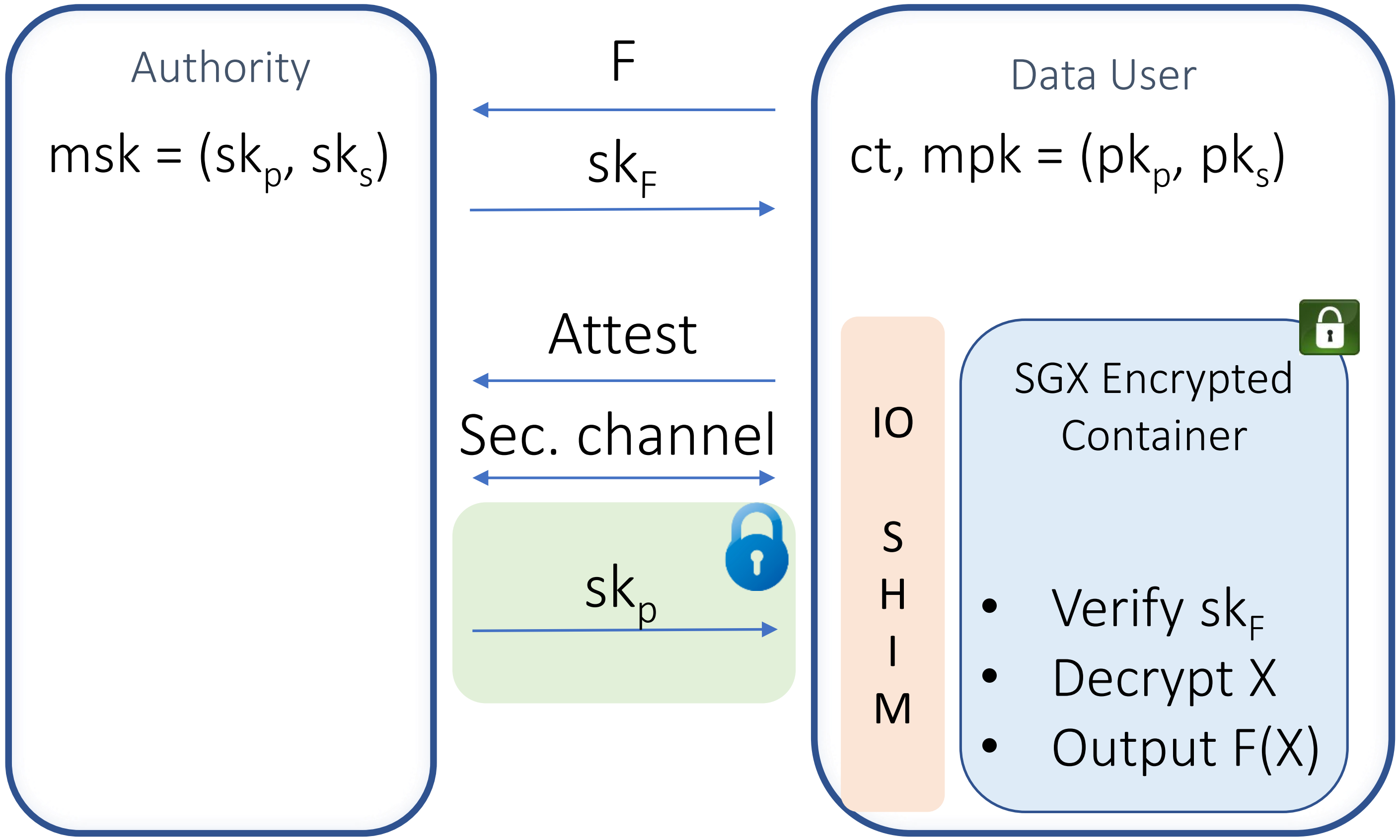
Problems:

1) Enc. container cannot talk over network?

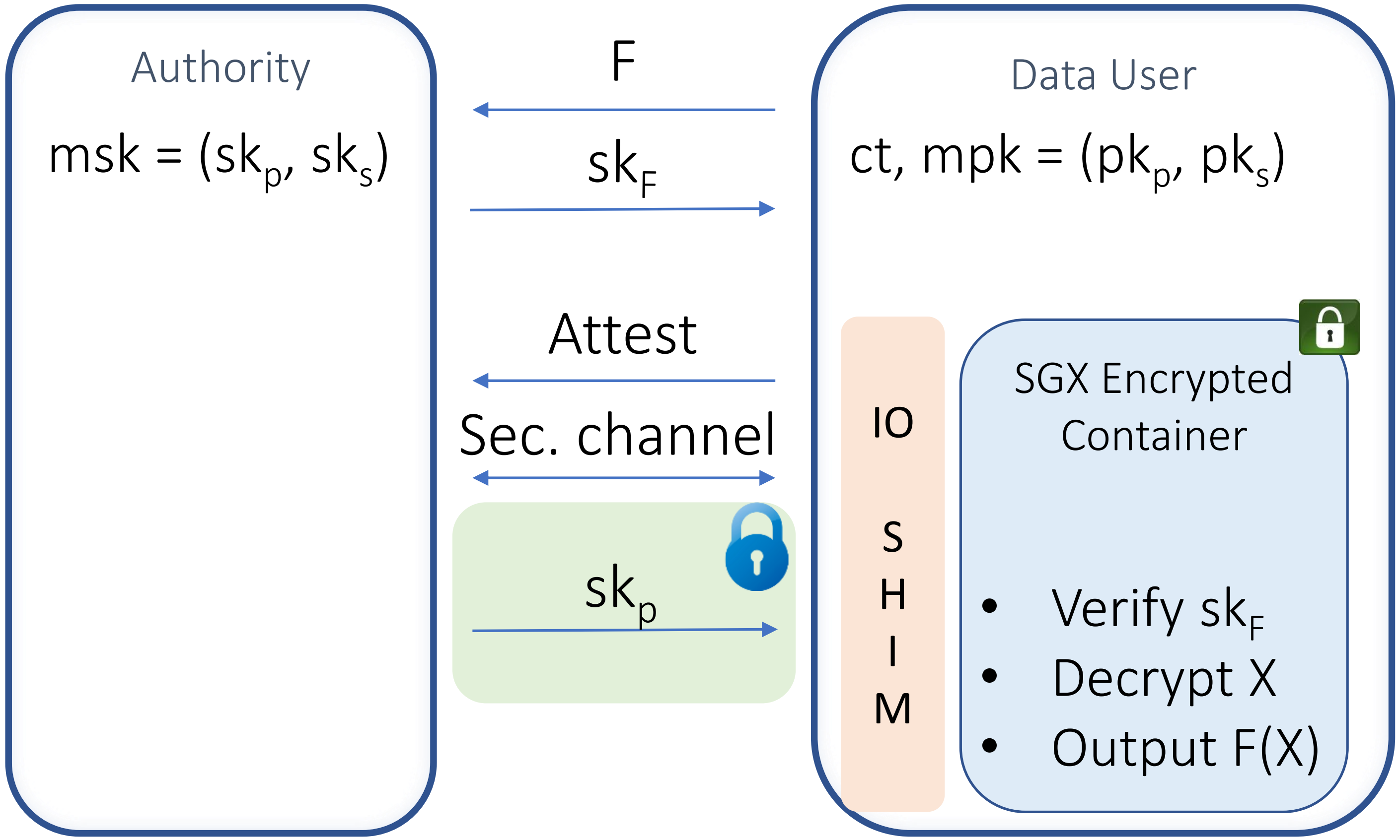
2) Which function to attest in enc. container?

$\mathrm{Dec}(sk_F, ct) \rightarrow F(X)$:

1) Enc. container cannot talk over network?



Dec(sk_F, ct) $\rightarrow$ F(X):



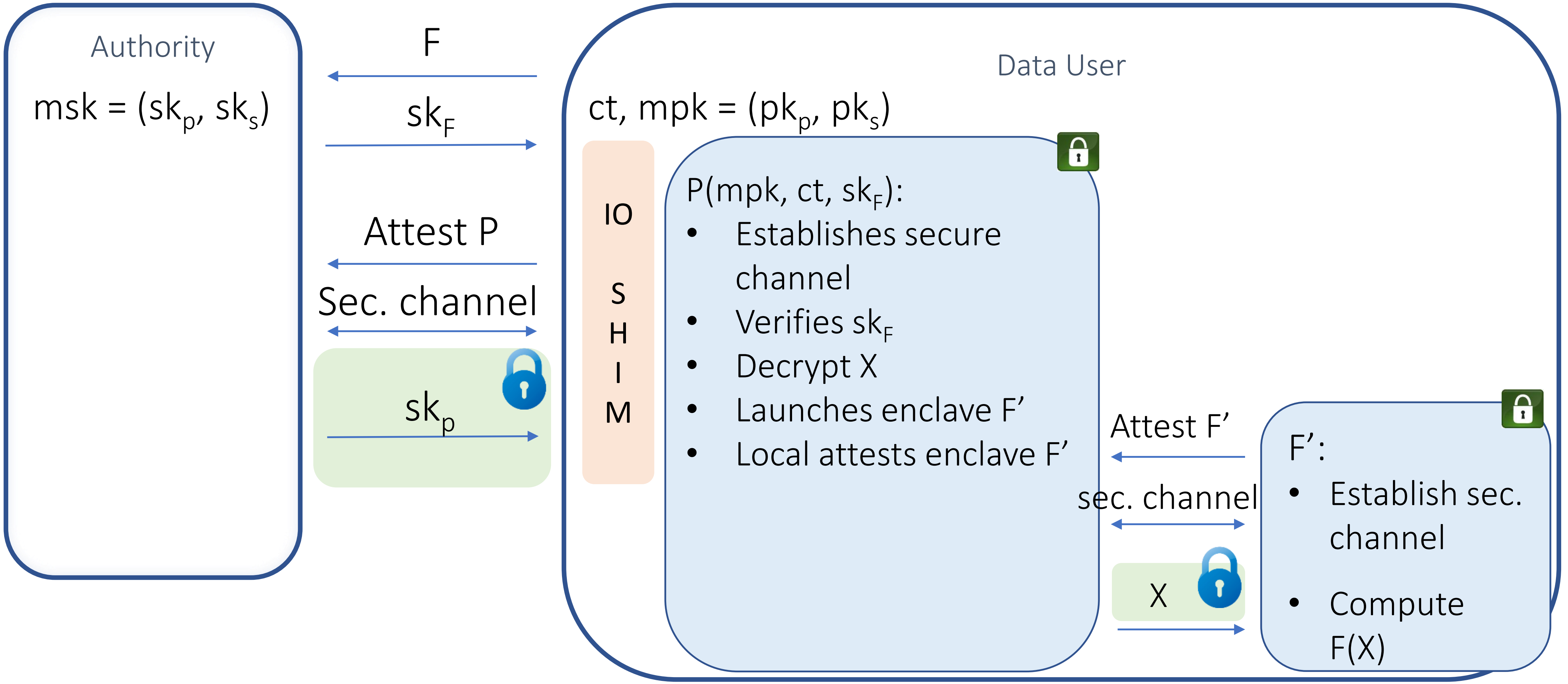
2) Which function to attest in enc. container?

Define: P(mpk, ct, sk_F):
1) Establish secure channel
2) Verify sk_F
3) Decrypt X
4) Output F(X)

Load and attest P

$\mathrm{Dec}(sk_F, ct) \rightarrow F(X)$:

2) Which function to attest in enc. container?



Q & A

Q: Adversary controls the IO Shim layer. Can she/he modify:

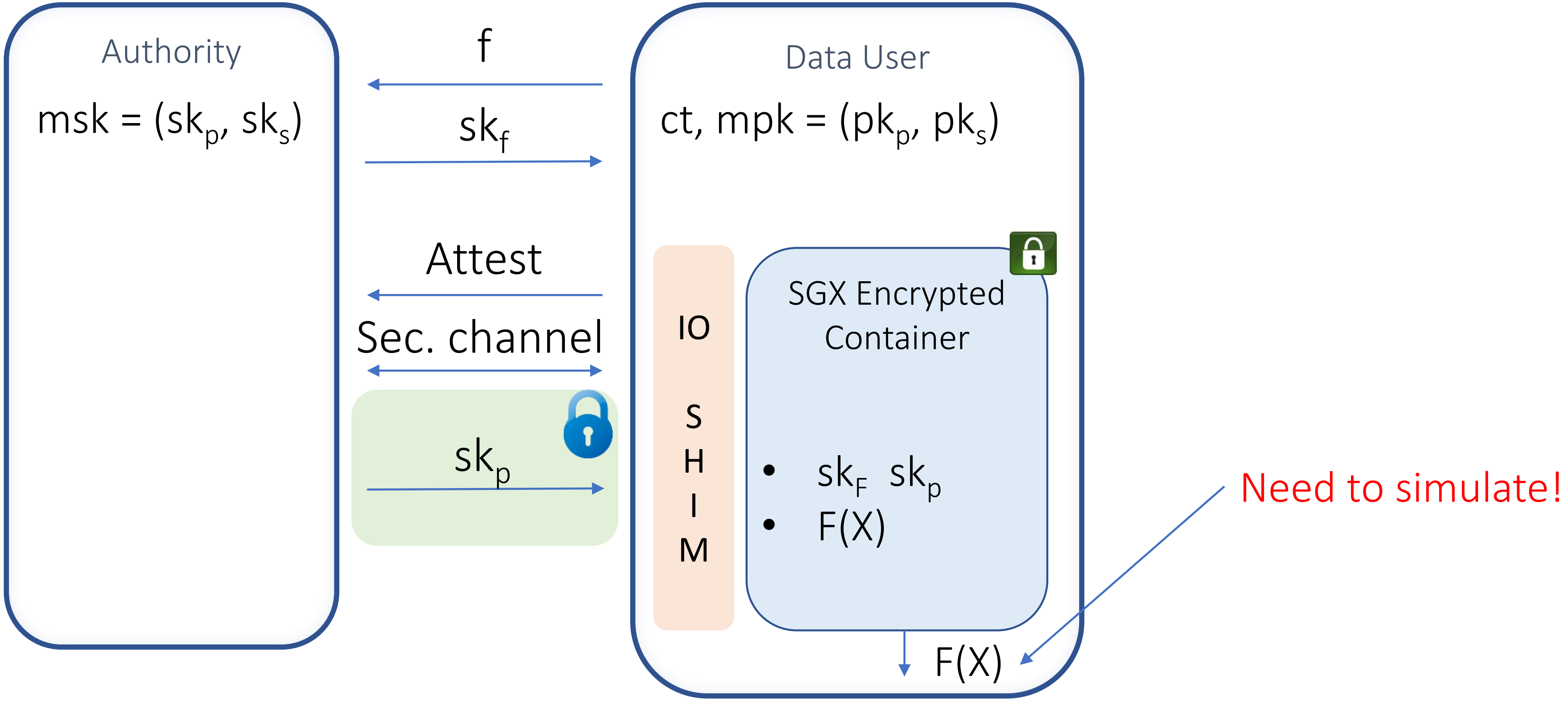
1. The secret key sk_F
2. Program loaded P
3. The encryption of the secret key sk_p and observe output F(X) to learn information about sk_p?

A:

1. No, follows by security of signature scheme
2. No, follows by attestation property of SGX
3. Channel must be protected with CCA2 properties.

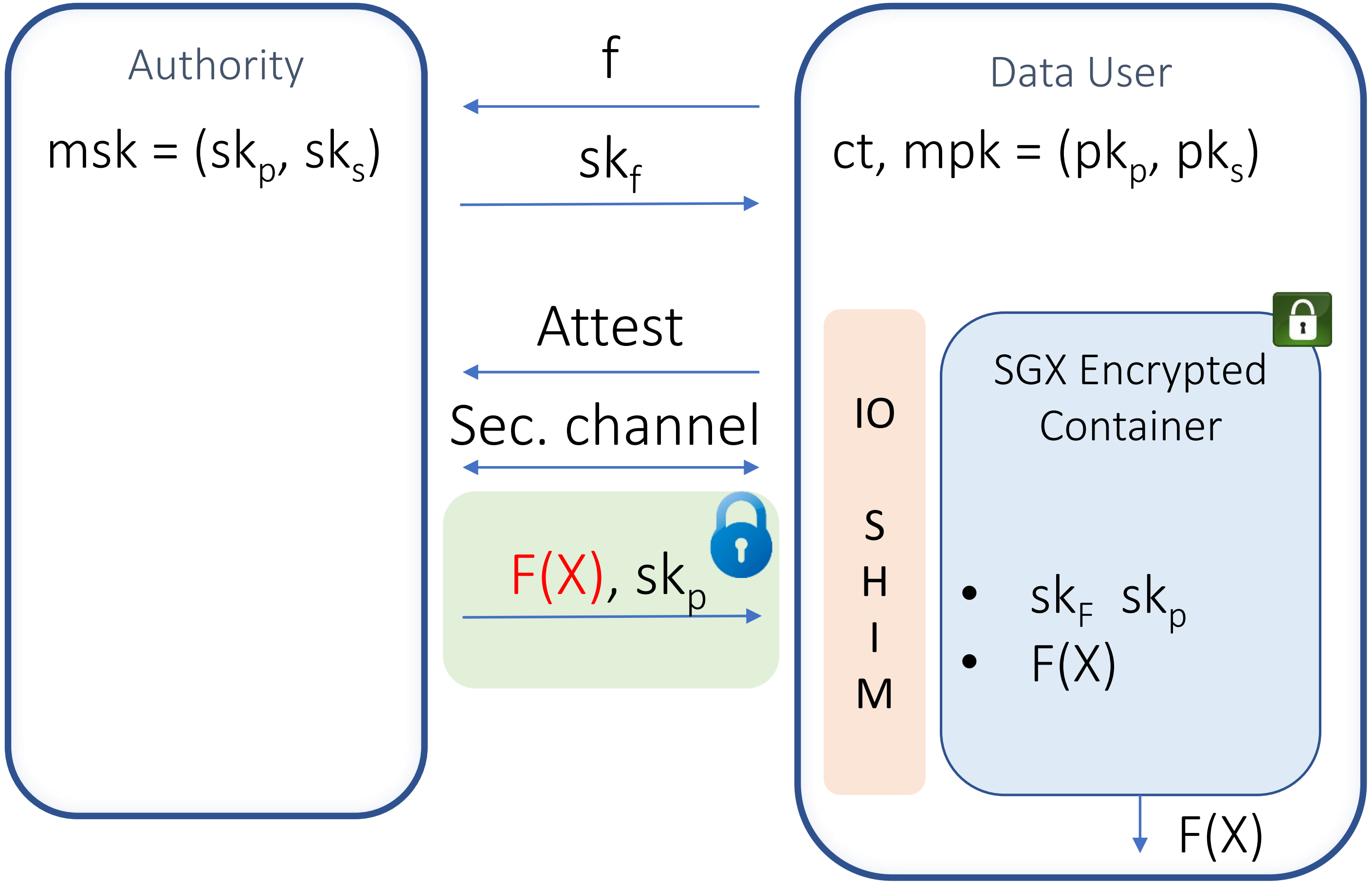
Q & A

Q: How does the proof work?



Q & A

Q: How does the proof work?



A:

- In simulation, F(X) comes from the authority via sec. channel (enc(0) in the real game)
- Indistinguishability of enc(0) and enc(F(X)) follows by sec. channel (not readily. need to use dual-encryption tech.)

Q & A

Q: What is “function description” and how does authority validate it?

A: An arbitrary C/C++ program code that is given to the authority. Authority can inspect the code, compile into sgx-enabled executable and sign the executable. sk_F = (executable, signature of the executable).

Q & A

Q: SGX is vulnerable to side-channels?

A: Yes, while inspecting the code of a function F, the authority can ensure that it side-channel free or augment it into such form before compiling.

Program P needs to be built side-channel free once and for all.

(Side-channel free: e.g., constant time.)

Q & A

Q: What happens if the data user restarts the node?

A: SGX has a mechanism to “seal” enclave secrets on persistent storage with a hardware-derived key.

Outline

- ✓ Motivation and our results
- ✓ Background on secure hardware (Intel SGX)
- ✓ Construction overview
- ✓ Proof insights
- Implementation details and performance

Implementation

Intel i5, 16 GB RAM, Intel SGX SDK 1.6 for Windows

Crypto Algorithms:

- PKE ElGamal (MSR_ECClib.lib) + AES-GCM
- Signature ECDSA (sgx_tcrypto.lib)

Supported functions

- Any function that can be loaded into an enclave
- And resist side-channels

Implementation

We implement *oblivious* IBE, ORE, 3-DNF, simple linear regression

By implementing data comparisons in registers, constant time, code-independent accesses [OSF+16]

- IBE : $ct \leftarrow Enc(ID, X)$ $X \leftarrow Dec(sk_{ID}, ct)$

- Order(X, Y) : Output 1 if $x > y$, else 0

- 3-DNF($\underbrace{X,Y,Z}_{\text{n-bit vectors}}$) : Output $(x_1 \wedge y_1 \wedge z_1) \vee \cdots \vee (x_n \wedge y_n \wedge z_n)$

- SimpLinReg($\{a_i, b_i\}$) : Output the best-fit (α, β) such that $b_i = \alpha + \beta\, a_i$

Evaluation

- FE.Setup : 130 ms (60 ms for KMEnclave creation)
- FE.KeyGen : 10 ms
- FE.Decrypt:

Functionality:	**IBE**	**ORE**	**3DNF**
create enclave	14.5 ms	20.7 ms	19.7 ms
local attest	1.6 ms	2.1 ms	2.1 ms
decrypt & eval	0.98 ms	0.84 ms	0.96 ms
Total	17.8 ms	23.78 ms	22.76 ms

Figure 5: Breakdown of FE.Decrypt run times for each of our IRON implementations of IBE, ORE, and 3DNF. The input in IBE consisted of a 3-byte tag and a 32-bit integer payload. The input pairs in ORE were 32-bit integers, and the input triplets in 3DNF were 16-bit binary strings. (The input types were chosen for consistency with the 5Gen experiments).

Evaluation

	$\mathbf{IBE}^{SGX}$	$\mathbf{IBE}^{[BF01]}$	$\times$ increase
|msg|:	35 bits	35 bits	NA
|c|:	175 bytes	471 bytes	2.69
decrypt:	17.8 ms	49 ms	2.75
decrypt*:	0.39 ms	49 ms	125.64

	$\mathbf{ORE}^{SGX}$	$\mathbf{ORE}^{5Gen}$	$\times$ increase
|msg| :	32 bits	32 bits	NA
|c| :	172 bytes	4.7 GB	$27.3 \cdot 10^6$
decrypt :	23.78 ms	4 m	$10.1 \cdot 10^3$
decrypt* :	0.32 ms	4 m	$750 \cdot 10^3$

	$\mathbf{3DNF}^{SGX}$	$\mathbf{3DNF}^{5Gen}$	$\times$ increase
|msg|:	16 bits	16 bits	NA
|c|:	170 bytes	2.5 GB	$14.7 \cdot 10^6$
decrypt:	22.76 ms	3 m	$7.9 \cdot 10^3$
decrypt*:	0.45 ms	3 m	$400 \cdot 10^3$

Figure 6: Comparison of decryption times and ciphertext sizes for the IRON implementation of IBE, ORE, 3DNF to cryptographic implementations. The 5Gen ORE and 3DNF implementation referenced here uses the CLT mmap with an 80-bit security parameter. The column decrypt gives the cost of running a single decryption, and decrypt* gives the amortized cost (per ciphertext tuple) of 10^3 decryptions.

Thank you!