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

ct = Enc(mpk, DB)

mpk, msk

NIH

National Institutes of Health

F1

F1(DB)

F2

F2(DB)

F3

F3(DB)

CT

sk_{F1}

sk_{F2}

sk_{F3}
FE Definition
[Boneh, Sahai, Waters 11]

\[(mpk, msk) \leftarrow \text{Setup}(1^n)\]  
\[ct \leftarrow \text{Enc}(mpk, X)\]  
\[sk_F \leftarrow \text{Keygen}(msk, F)\]  
\[F(X) \leftarrow \text{Dec}(sk_F, ct)\]

Authority (NIH)  
Data Owner (may not be NIH)  
Authority  
Service / Data User
FE Security - Informal

- **Simulation (SIM):**

  Adversary given \((\text{sk}_{F_1}, \text{sk}_{F_2}, \ldots, \text{sk}_{F_q})\) and \(\text{Enc}(\text{mpk}, X)\), learns only \(F_1(X), F_2(X), \ldots, F_q(X)\)

- **Indistinguishability (IND):**

  Adversary given access to \((\text{sk}_{F_1}, \text{sk}_{F_2}, \ldots, \text{sk}_{F_q})\), cannot distinguish between \(\text{Enc}(\text{mpk}, X_0)\) and \(\text{Enc}(\text{mpk}, X_1)\) where \(F_i(X_0) = F_i(X_1)\) for all \(i\).
FE Security – semi-formal

[BSW11, O’N10]

\[ (X, st) \xrightarrow{c \approx} (X, st) \]

Real World

Ideal World

(FE scheme)

Adv

F1, F2, ...

SK_{F1}, SK_{F2}, ...

X

ct

MPK

Adv

F1, F2, ...

SK_{F1}, SK_{F2}, ...

F1(X), F2(X), ...

ct

MPK

Sim
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).

Untrusted Host

Container:
- Program code
- Stack
- Libraries
- Internal states
- Data pages

-memory

User program/data

User program/data

CPU

Only the CPU is tamper safe from the adversary

Standard CPU Logic + Hardware Module + Encryption Routines (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 \(\text{wrt}\) 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

Proof

\[ \text{Proof}_P = \text{Sign}(sk, H(P)) \]

User program P

User program/data

Attest(pk, P, Proof_P)

CPU

pk, sk

pk

Proof_P

User program P

Proof_P

User program/data

User program/data
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

Proof

\[ \text{Proof}_{P(X)} = \text{Sign}(sk, P(X)) \]

User program P

Input X

pk

P(X), Proof_{P(X)}

Verify(pk, P(X), Proof_{P(X)})

pk, sk

CPU

User program/data
SGX Formal Algorithms

- Setup($1^n$) $\rightarrow$ (sk, pk)
- Load_{sk}(P) $\rightarrow$ Proof_p
- Attest(pk, P, Proof_p) $\rightarrow$ 0/1
- Run_{sk}(X) $\rightarrow$ (P(X), Proof_{P(X)})
- Verify(pk, P(X), Proof_{P(X)}) $\rightarrow$ 0/1
SGX Initialization and Runtime

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

1. **Load($P$)**: Load the program $P$ into memory.
2. **Proof$_P$**: Generate a proof $Proof_P$ for the program.$P$.
3. **Attest($pk, P, Proof_P$)**: Attest to the authenticity of the program.
4. **Sec. channel**: Secure channel for communication.
5. **X**: Input data $X$.
6. **P($X$), Proof$_{P(X)}$**: Compute $P(X)$ and generate a proof $Proof_{P(X)}$.
7. **Verify($pk, P(X), Proof_{P(X)}$)**: Verify the correctness and integrity of the computation result.

The diagram illustrates the secure computation workflow within the secure enclaves provided by Intel SGX technology.
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

Disk encryption
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) → ct

Setup(1^k) → (mpk, msk)
1) s.setup(1^k) → (vk_s, sk_s)
2) p.setup(1^k) → (pk_p, sk_p)
3) mpk = (pk_p, vk_s), msk = (sk_p, sk_s)

Dec(sk_F, ct) → F(X)
(next slide)

Keygen(msk, F) → sk_F
1) s.sign(sk_s, F) → sk_F
Dec\( (sk_F, ct) \rightarrow F(X) \):
Dec($sk_F$, ct) → $F(X)$:

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 $P$. 

Dec\(sk_F, ct\) → \(F(X)\):

2) Which function to attest in enc. container?

Authority
msk = \((sk_p, sk_s)\)

F

\(sk_F\)

\(sk_p\)

Attest P
Sec. channel

Data User
ct, mpk = \((pk_p, pk_s)\)
P(mpk, ct, sk_F):
• Establishes secure channel
• Verifies \(sk_F\)
• Decrypt X
• Launches enclave \(F'\)
• Local attests enclave \(F'\)

\(X\)

Attest \(F'\)
sec. channel

\(F'\):
• Establish sec. channel
• Compute \(F(X)\)
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: How does the proof work?

Authority

\[ msk = (sk_p, sk_s) \]

Data User

\[ ct, mpk = (pk_p, pk_s) \]

\[ f \]

\[ sk_f \]

Need to simulate!
Q: How does the proof work?

A:
- In simulation, $F(X)$ comes from the authority via sec. channel ($\text{enc}(0)$ in the real game)
- Indistinguishability of $\text{enc}(0)$ and $\text{enc}(F(X))$ 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. \( sk_F = (\text{executable}, \text{signature of the executable}) \).
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: 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**
  
  : $\text{ct} \leftarrow \text{Enc}(\text{ID}, X)$
  
  $X \leftarrow \text{Dec}(sk_{\text{ID}}, \text{ct})$

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

- **3-DNF($X,Y,Z$)**
  
  : Output $(x_1 \land y_1 \land z_1) \lor \cdots \lor (x_n \land y_n \land z_n)$

  - n-bit vectors

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

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

<table>
<thead>
<tr>
<th>Functionality</th>
<th>IBE</th>
<th>ORE</th>
<th>3DNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>create enclave</td>
<td>14.5 ms</td>
<td>20.7 ms</td>
<td>19.7 ms</td>
</tr>
<tr>
<td>local attest</td>
<td>1.6 ms</td>
<td>2.1 ms</td>
<td>2.1 ms</td>
</tr>
<tr>
<td>decrypt &amp; eval</td>
<td>0.98 ms</td>
<td>0.84 ms</td>
<td>0.96 ms</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>17.8 ms</td>
<td>23.78 ms</td>
<td>22.76 ms</td>
</tr>
</tbody>
</table>

**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).
Table 1: Comparison of decryption times and ciphertext sizes for the IROK 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.

<table>
<thead>
<tr>
<th></th>
<th>IBE^{SGX}</th>
<th>IBE^{BF01}</th>
<th>× increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>msg</td>
<td>35 bits</td>
<td>35 bits</td>
</tr>
<tr>
<td>c</td>
<td>175 bytes</td>
<td>471 bytes</td>
<td>2.69</td>
</tr>
<tr>
<td>decrypt</td>
<td>17.8 ms</td>
<td>49 ms</td>
<td>2.75</td>
</tr>
<tr>
<td>decrypt*</td>
<td>0.39 ms</td>
<td>49 ms</td>
<td>125.64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ORE^{SGX}</th>
<th>ORE^{5Gen}</th>
<th>× increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>msg</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>c</td>
<td>172 bytes</td>
<td>4.7 GB</td>
<td>27.3 \times 10^6</td>
</tr>
<tr>
<td>decrypt</td>
<td>23.78 ms</td>
<td>4 m</td>
<td>10.1 \times 10^3</td>
</tr>
<tr>
<td>decrypt*</td>
<td>0.32 ms</td>
<td>4 m</td>
<td>750 \times 10^3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>3DNF^{SGX}</th>
<th>3DNF^{5Gen}</th>
<th>× increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>msg</td>
<td>16 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>c</td>
<td>170 bytes</td>
<td>2.5 GB</td>
<td>14.7 \times 10^6</td>
</tr>
<tr>
<td>decrypt</td>
<td>22.76 ms</td>
<td>3 m</td>
<td>7.9 \times 10^3</td>
</tr>
<tr>
<td>decrypt*</td>
<td>0.45 ms</td>
<td>3 m</td>
<td>400 \times 10^3</td>
</tr>
</tbody>
</table>
Thank you!