ICE: A Passive, High-Speed, State-Continuity Scheme

Raoul Strackx

KU Leuven

December 10, 2014
Outline

1. Introduction
2. What’s still missing: State Continuity
3. Our Solution: ICE
4. Evaluation
5. Conclusion
What we have...

- Apps
- OS
- TPM
- CPU
- Mem
- HDD

- Trusted
- Untrusted
What we have...

- Windows 7: 40 million LoC
- Linux kernel: 12 million LoC
- Firefox: 6 million LoC
What we have...

- More Secure Libraries
- Harden Legacy Software Automatically
- Memory-Safe Languages
- Software Verification
- OS Protection Features
- Hardware Security Modules
What we have...

- More Secure Libraries
- Harden Legacy Software Automatically
- Memory-Safe Languages
- Software Verification
- OS Protection Features
- Hardware Security Modules

...But software security is still a mess
PMAs: A possible way out

Main idea:
- Carve out sensitive parts of an application
- Apply hardware-based isolation to these modules
- Modules have complete control over their own security

This idea is known as a Protected-Module Architecture and will soon be available on commodity Intel processors (Intel SGX)
An example:
The result:

- **App**
- **App**
- **App**
- **App**
- **OS**
- **TPM**
- **CPU**
- **Mem**
- **HDD**

- Trusted
- Untrusted

SGX enclaves
What we have...

- Verification
- Fully Abstract Compilation
- Isolation
- SRNG
- Key deriv.
- TPM
- NVRAM

Source code verification
Secure mod. Compilation
HW reqs.
What we can guarantee:

- SW properties of modules cannot be broken by SW attacks (Fully-abstract compilation\(^1\) & verification\(^2\))
- Resistance against powerful HW attacks

\(^1\)Patrignani et al. “Secure compilation to Protected Module Architectures”. Accepted for publication in Transactions on Programming Languages and Systems (TOPLAS)

\(^2\)Agten, Jacobs, and Piessens. “Sound Modular Verification of C Code Executing in an Unverified Context”. 2015. Accepted for publication in Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’15)
What we can guarantee:

- SW properties of modules cannot be broken by SW attacks (Fully-abstract compilation\(^1\) & verification\(^2\))
- Resistance against powerful HW attacks

While the system executes continuously

... but in real-life systems crash, reboot and power goes down

\(^1\) Patrignani et al. “Secure compilation to Protected Module Architectures”. Accepted for publication in Transactions on Programming Languages and Systems (TOPLAS)

\(^2\) Agten, Jacobs, and Piessens. “Sound Modular Verification of C Code Executing in an Unverified Context”. 2015. Accepted for publication in Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’15)
Outline

1. Introduction
2. What’s still missing: State Continuity
3. Our Solution: ICE
4. Evaluation
5. Conclusion
Providing State Continuity

Password-checking module

```java
static int tries_left = 3;
static string password = "mypassword";

boolean check_password( string input ) {
    if ( tries_left == 0 )
        return false;
    if ( password == input ) {
        tries_left = 3;
        store_state() //store state encrypted and signed
        return true;
    } else {
        --tries_left;
        store_state() //store state encrypted and signed
        return false;
    }
}
```
What's still missing: State Continuity

Providing State Continuity

Password-checking module

```java
static int tries_left = 3;
static string password = "mypassword";

boolean check_password( string input ) {
    if ( tries_left == 0 )
        return false;

    if ( password == input ) {
        tries_left = 3;
        store_state() //store state encrypted and signed
        return true;
    } else {
        --tries_left;
        store_state() //store state encrypted and signed
        return false;
    }
}
```
Consider two possible attacks:

**Attack 1:** Providing stale state
- Store the initial state
- Guess the password 3 times
- Crash the system
- Provide the initial state stored on disk as being fresh

```java
static int tries_left = 3;
static string password = "mypassword";

boolean check_password( string input ) {
    if ( tries_left == 0 )
        return false;
    if ( password == input ) {
        tries_left = 3;
        store_state()
        return true;
    } else {
        --tries_left;
        store_state()
        return false;
    }
}
```
Providing State Continuity

**Attack 2: Timing attack**
- Guess the password
- Crash the system before storing the state if:
  
  \[ \text{time}_{\text{correct}} < \text{time}_{\text{wrong}} \]

- Restore the last state

Password-checking module

```java
static int tries_left = 3;
static string password = "mypassword";

boolean check_password( string input ) {
    if ( tries_left == 0 )
        return false;

    if ( password == input ) {
        tries_left = 3;
        store_state()
        return true;
    }
    else {
        --tries_left;
        store_state()
        return false;
    }
}
```
Requirements

We need to add support for *state continuity*

Three key properties:

- **Safety**: prevent the use of stale data
- **Liveness**: a crash of the machine must not leave the machine unable to advance
- **Speed**: must be applicable to most applications
Outline

1. Introduction
2. What’s still missing: State Continuity
3. Our Solution: ICE
4. Evaluation
5. Conclusion
Side note:

- First store input, then process
- Modules are deterministic
  
  → We only need to consider state-continuous storage
Side note:

- First store input, then process
- Modules are deterministic
- We only need to consider state-continuous storage
- (Only provide state-continuous execution to a single module)
Building blocks: Guards

Definition:

\[ \text{guard}_i(n) = (\text{Hash}^i(n), i) \]

Incrementing a guard:

\[ \text{guard}_0(n) = (n, 0) \]
\[ \text{guard}_i(n) = (\text{Hash}^i(n), i) \]
\[ \text{guard}_{i+1}(n) = (\text{Hash}^{i+1}(n), i + 1) \]

Comparison of guards:

\[ (n, i) \leq (m, j) \iff \begin{cases} 
n = m & \text{if } i = j \\
(\text{Hash}(n), i + 1) \leq (m, j) & \text{if } i < j 
\end{cases} \]
Building Blocks: Cubes

State stored on disk = Cubes:

State guard MAC
encrypted
“Guarded Memory”
- Non-volatile memory
- Contents remains secret *only* while power is on
- Power-off will make the contents world-readable
- Stores freshness information
Main Idea

- Enclaves keep their state between invokations
- Store a guard that acts as a “pointer” to the fresh cube in guarded memory
- This pointer only leaks on crash/reboot/loss of power
- As no previous guard leaked, knowledge of this pointer proves freshness of the cube
Init Step

```
initial_state( State s0 )

gmem.enable_protection();
guard = gen_guard();
key = gen_key();
hdd.write( new cube( s0, guard, key ) );
gmem.write( guard );
nvram.nvwrite( key, guard );
```
**Init Step**

```
initial_state( State s0 )

gmem.enable_protection();
guard = gen_guard();
key = gen_key();
hdd.write( new cube( s0, guard, key ) );
gmem.write( guard );
nvram.nvwrite( key, guard );
```
Our Solution: ICE

Init Step

```java
initial_state( State s0 )

gmem.enable_protection();
guard = gen_guard();
key = gen_key();
hdd.write( new cube( s0, guard, key ) );
gmem.write( guard );
nvram.nvwrite( key, guard );
```
Our Solution: ICE

Init Step

```
initial_state( State s0 )

gmem.enable_protection();
guard = gen_guard();
key = gen_key();
hdd.write( new cube( s0, guard, key ) );
gmem.write( guard );
nvram.nvwrite( key, guard );
```
Our Solution: ICE

Init Step

```
initial_state( State s0 )

gmem.enable_protection();
guard = gen_guard();
key = gen_key();
hdd.write( new cube( s0, guard, key ) );
gmem.write( guard );
nvram.nvwrite( key, guard );
```
store_state(State s1)

++guard;
hdd.write(new cube(s1, guard, key));
gmem.write(guard);
Our Solution: ICE

Next Step

store_state(State s1)

```java
++guard;
hdd.write( new cube( s1, guard, key ) );
gmem.write( guard );
```
store_state( State s2 )

++guard;
hdd.write( new cube( s2, guard, key ) );
gmem.write( guard );
store_state( State s2 )

++guard;
hdd.write( new cube( s2, guard, key ) );
gmem.write( guard );
store_state( State s3 )

++guard;
hdd.write( new cube( s3, guard, key ) );
gmem.write( guard );
store_state( State s3 )

++guard;
hdd.write( new cube( s3, guard, key ) );
gmem.write( guard );
recovery_step( void )

...  
bool is_fresh( Cube *cube, Guard fresh ) {  
    return ( check_mac( cube, tpm.keys ) &&  
        fresh.value == cube->guard.value &&  
        tpm.guard <= cube->guard );  
}
Our Solution: ICE

Crash!

recovery_step( void )

...  
bool is_fresh( Cube *cube, Guard fresh ) {
    return ( check_mac( cube, tpm.keys )
             && fresh.value == cube->guard.value
             && tpm.guard <= cube->guard );
}
recovery_step( void )

... 
bool is_fresh( Cube *cube, Guard fresh ) {
    return ( check_mac( cube, tpm.keys ) && 
             fresh.value == cube->guard.value && 
             tpm.guard <= cube->guard );
}
Outline

1. Introduction
2. What’s still missing: State Continuity
3. Our Solution: ICE
4. Evaluation
5. Conclusion
Evaluation: Security

- System can do down at *any* point in time
- Formal + machine checked proof
- (More details: see paper/talk to me offline)
Evaluation: Performance

With the right hardware support:
587 times faster than TPM NVRAM accesses!

<table>
<thead>
<tr>
<th></th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSD (in ms)</strong></td>
<td>-lice0</td>
</tr>
<tr>
<td>computation</td>
<td>0.06</td>
</tr>
<tr>
<td>writing guard</td>
<td>0.33</td>
</tr>
<tr>
<td>writing cubes</td>
<td>14.61</td>
</tr>
<tr>
<td>total</td>
<td>15.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HDD (in ms)</strong></td>
<td>-lice0</td>
</tr>
<tr>
<td>computation</td>
<td>0.06</td>
</tr>
<tr>
<td>writing guard</td>
<td>0.35</td>
</tr>
<tr>
<td>writing cubes</td>
<td>112.80</td>
</tr>
<tr>
<td>total</td>
<td>113.21</td>
</tr>
</tbody>
</table>
Why is this important:

- It’s applicable in practice *without* reducing SGX’s attack model
- Speed of TPM NVRAM is *not* an issue
- Opens up the way to rely on even less HW requirements
- Enables a large range of applications

---

Provide state-continuous execution that is

- A practical security primitive for PMAs
- Proven secure
- Fast
- Very limited HW requirements
Questions?

raoul.strackx@cs.kuleuven.be
https://distrinet.cs.kuleuven.be/software/pcbac