INTEL® SOFTWARE GUARD EXTENSIONS (INTEL® SGX)

Intel Corporation
June 2015
Reference Number: 332680-002
Revision Number: 1.1

Legal Disclaimer

Notice: This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at Intel.com, or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo and SGX are trademarks of Intel Corporation in the U. S. and/or other countries.

*Other names and brands may be claimed as the property of others.

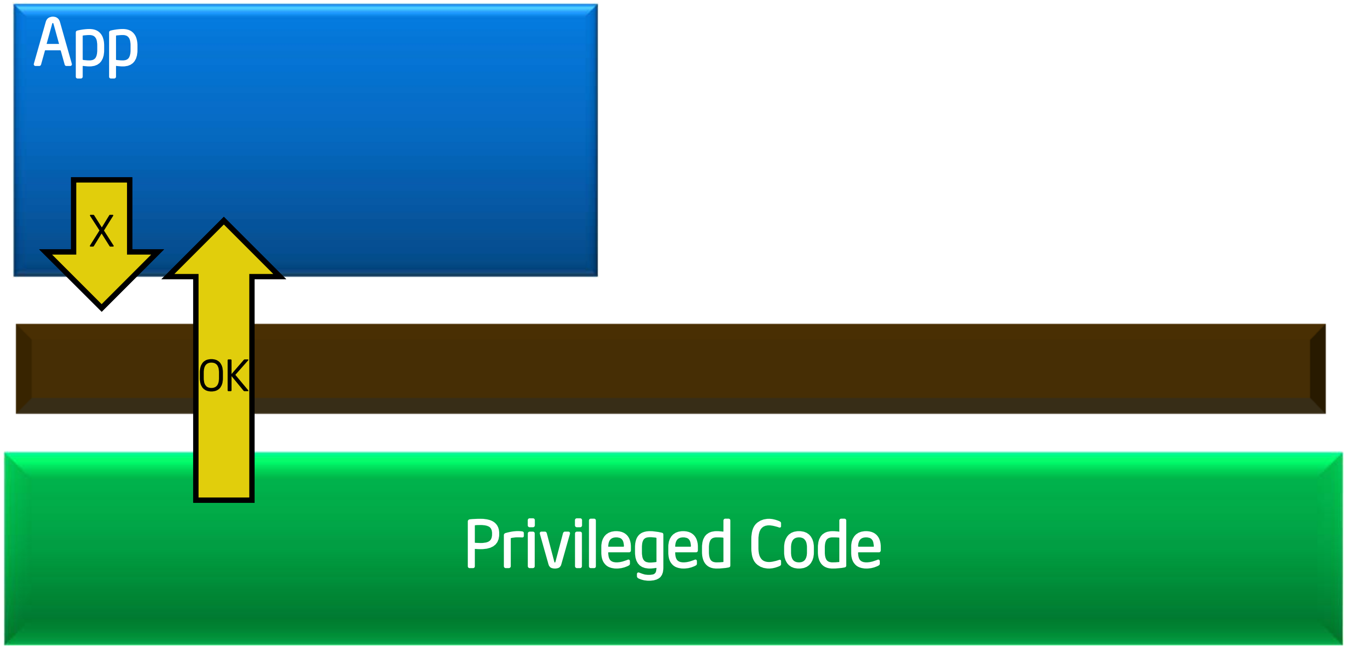
Agenda

- Overview
- In depth architecture
- SGX key hierarchy, attestation, provisioning
- SGX Security Properties
- Side Channel Evaluation Tool Demo
- Formal Validation and Modeling
- Memory Encryption Engine Architecture and Security Properties
- Implementation of SGX on Intel Core Processors
- Software: run time environment, EPC management, and SDK for Linux

Introduction

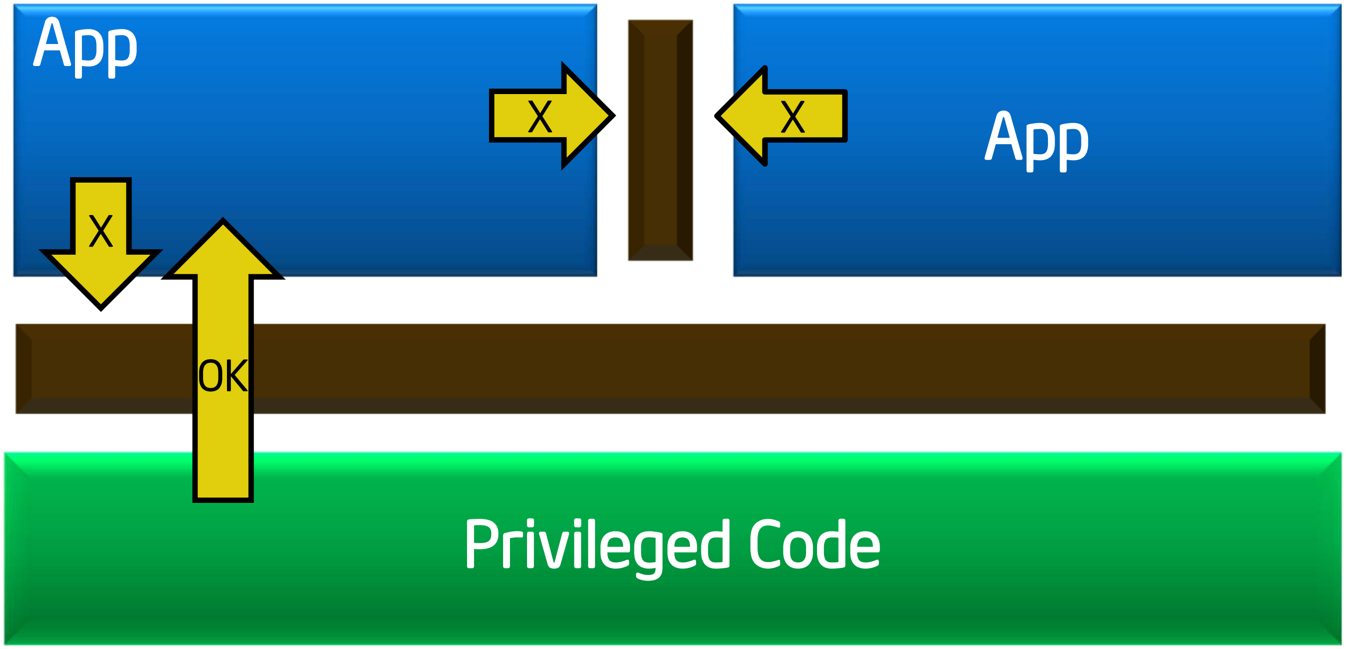
The Basic Issue: Why Aren't Compute Devices Trustworthy?

Protected Mode (rings) protects OS from apps …



The Basic Issue: Why Aren't Compute Devices Trustworthy?

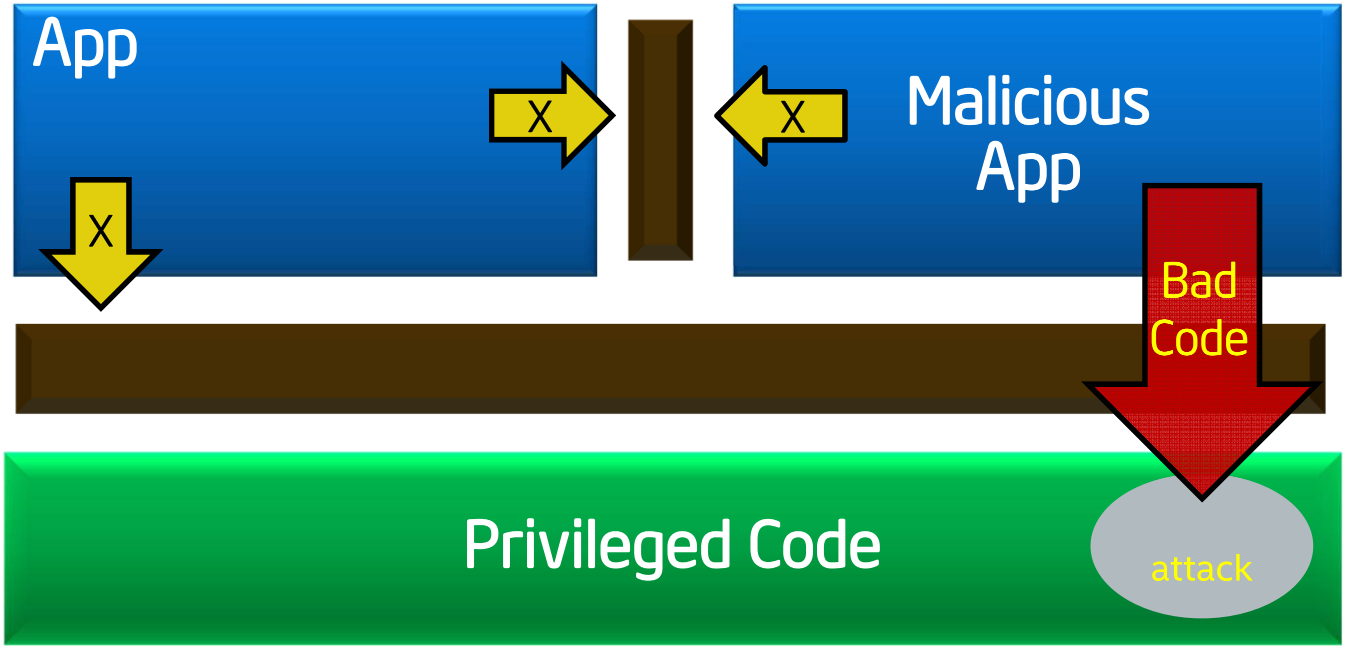
Protected Mode (rings) protects OS from apps …



… and apps from each other …

The Basic Issue: Why Aren't Compute Devices Trustworthy?

Protected Mode (rings) protects OS from apps …



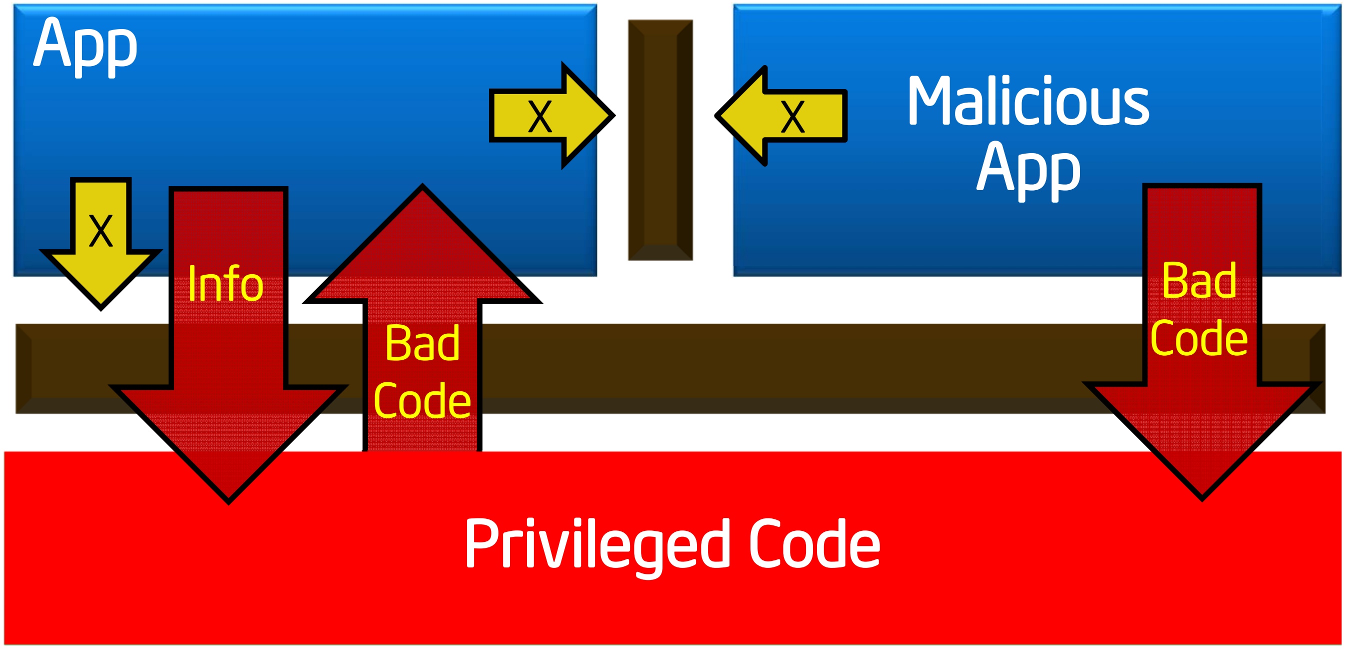
… and apps from each other …

… UNTIL a malicious app exploits a flaw to gain full privileges and then tampers with the OS or other apps

Apps not protected from privileged code attacks

The Basic Issue: Why Aren’t Compute Devices Trustworthy?

Protected Mode (rings) protects OS from apps …



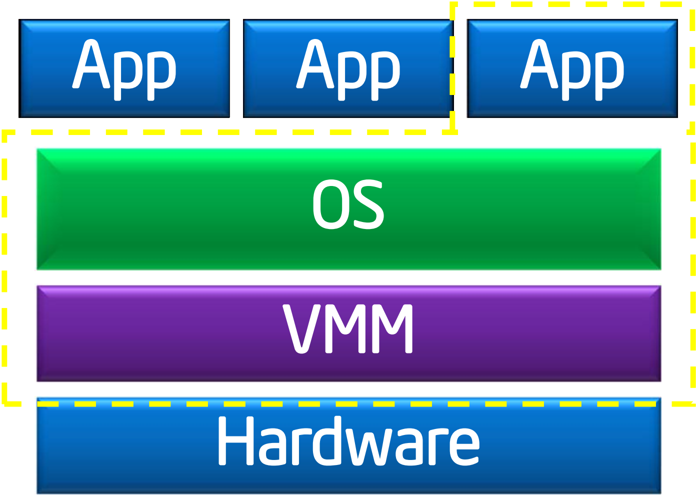
… and apps from each other …

… UNTIL a malicious app exploits a flaw to gain full privileges and then tampers with the OS or other apps

Apps not protected from privileged code attacks

Reduced attack surface with SGX

Attack surface today



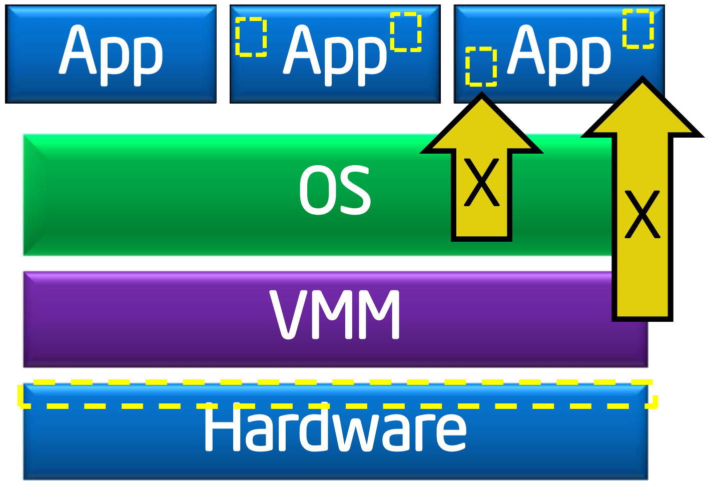
Attack Surface ⬚

Reduced attack surface with SGX

Application gains ability to defend its own secrets

- Smallest attack surface (App + processor)
- Malware that subverts OS/VMM, BIOS, Drivers etc. cannot steal app secrets

Attack surface with Enclaves



Attack Surface ⬚

Reduced attack surface with SGX

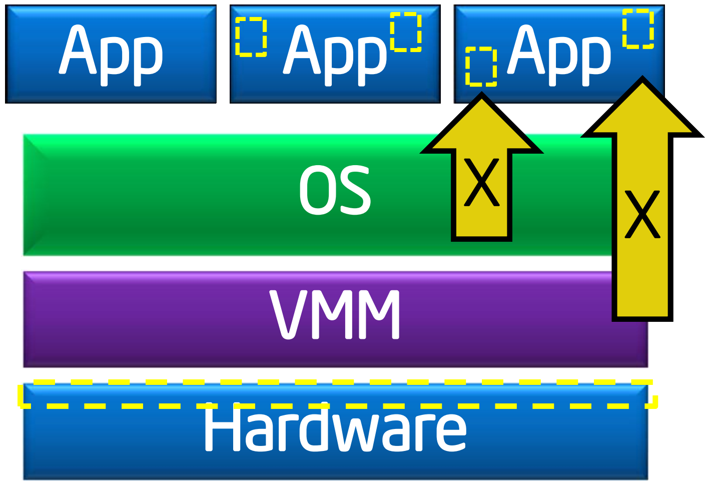
Application gains ability to defend its own secrets

- Smallest attack surface (App + processor)
- Malware that subverts OS/VMM, BIOS, Drivers etc. cannot steal app secrets

Familiar development/debug

- Single application environment
- Build on existing ecosystem expertise

Attack surface with Enclaves



Attack Surface ⬚

Reduced attack surface with SGX

Application gains ability to defend its own secrets

- Smallest attack surface (App + processor)
- Malware that subverts OS/VMM, BIOS, Drivers etc. cannot steal app secrets

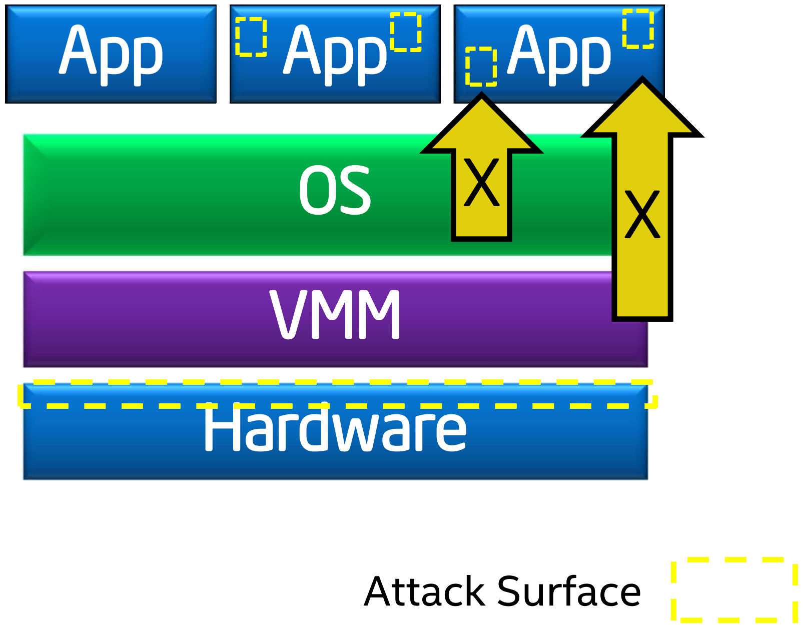
Familiar development/debug

- Single application environment
- Build on existing ecosystem expertise

Familiar deployment model

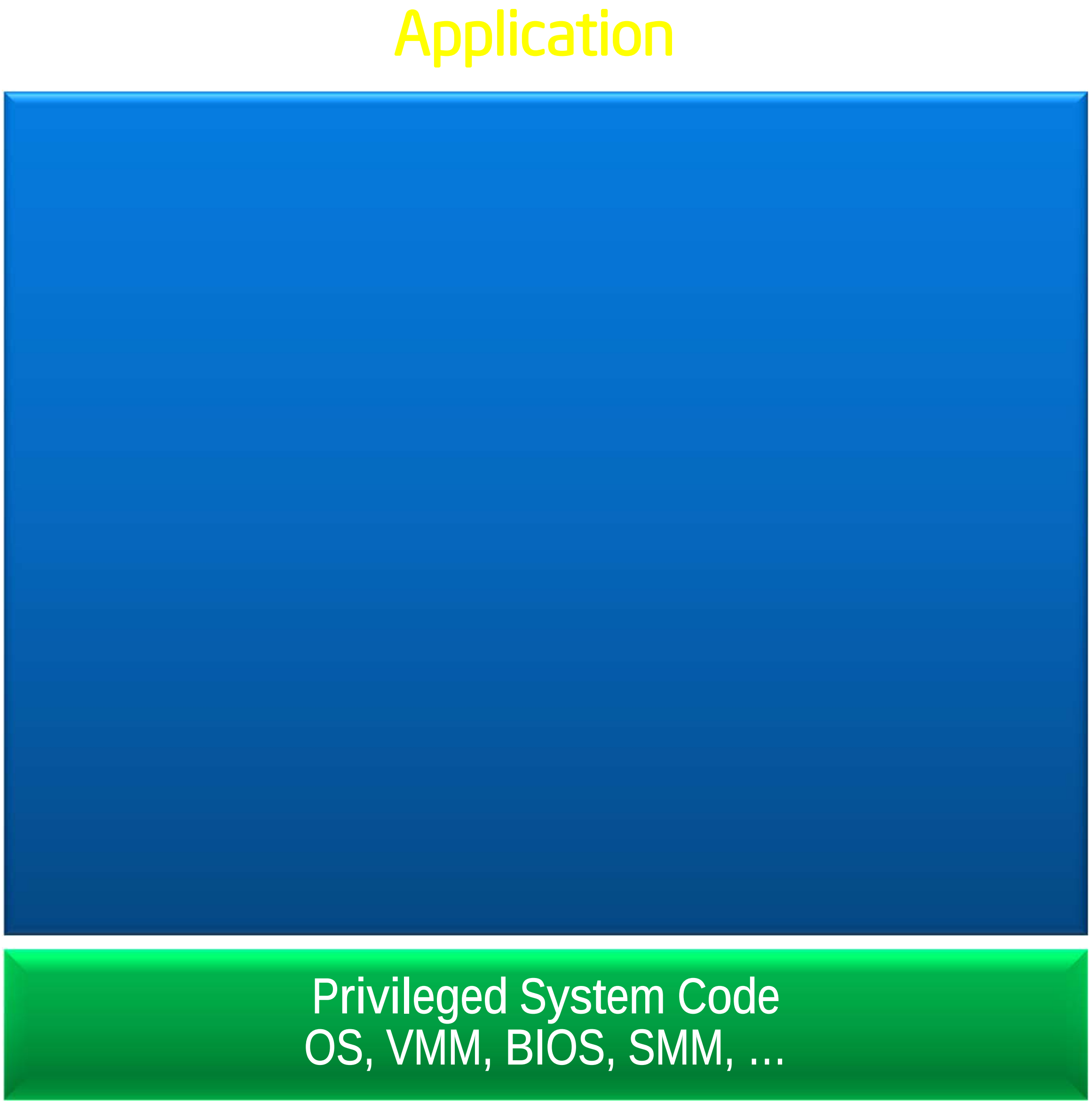
- Platform integration not a bottleneck to deployment of trusted apps

Attack surface with Enclaves



Scalable security within mainstream environment

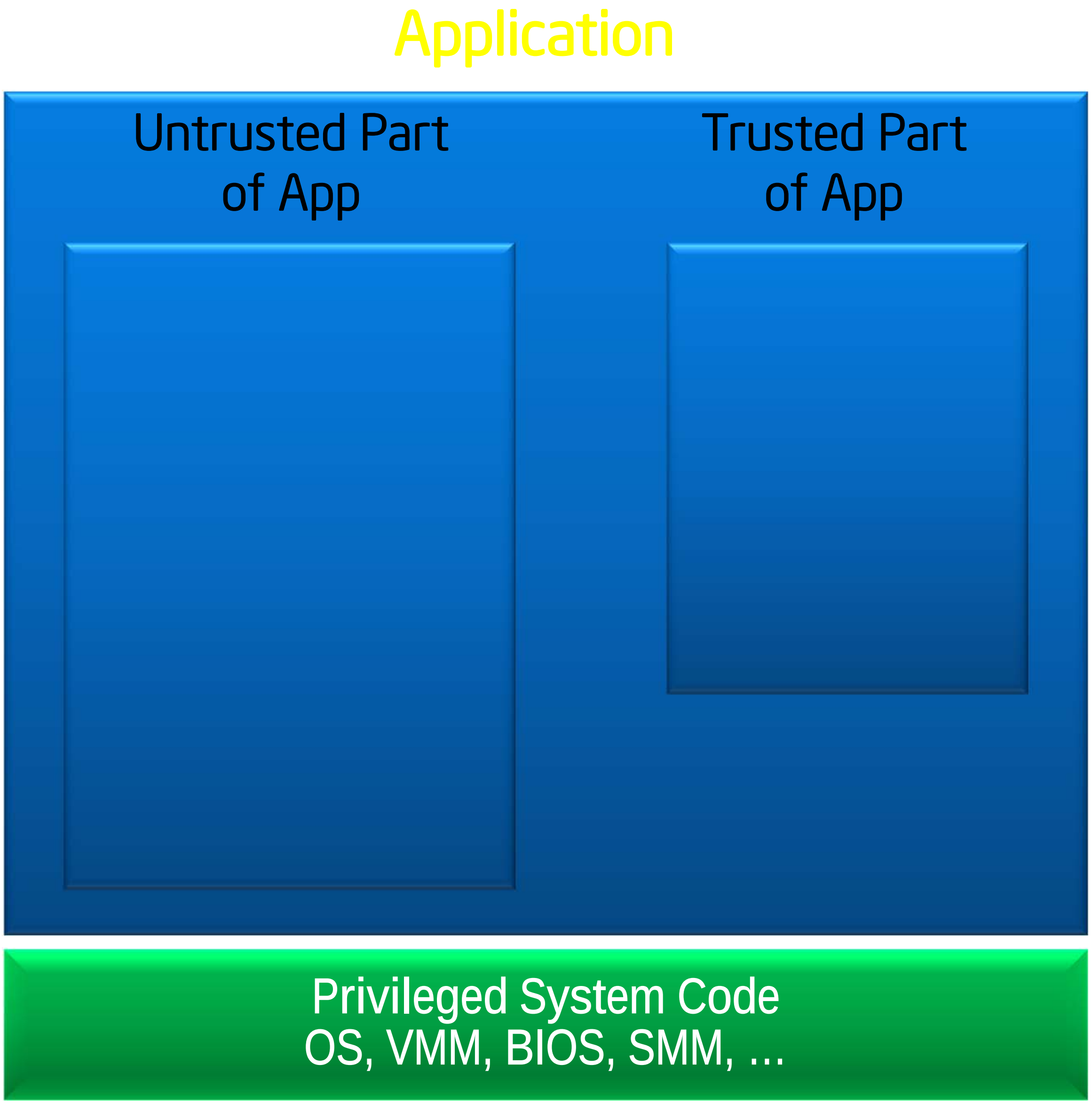
How SE Works: Protection vs. Software Attack



Application

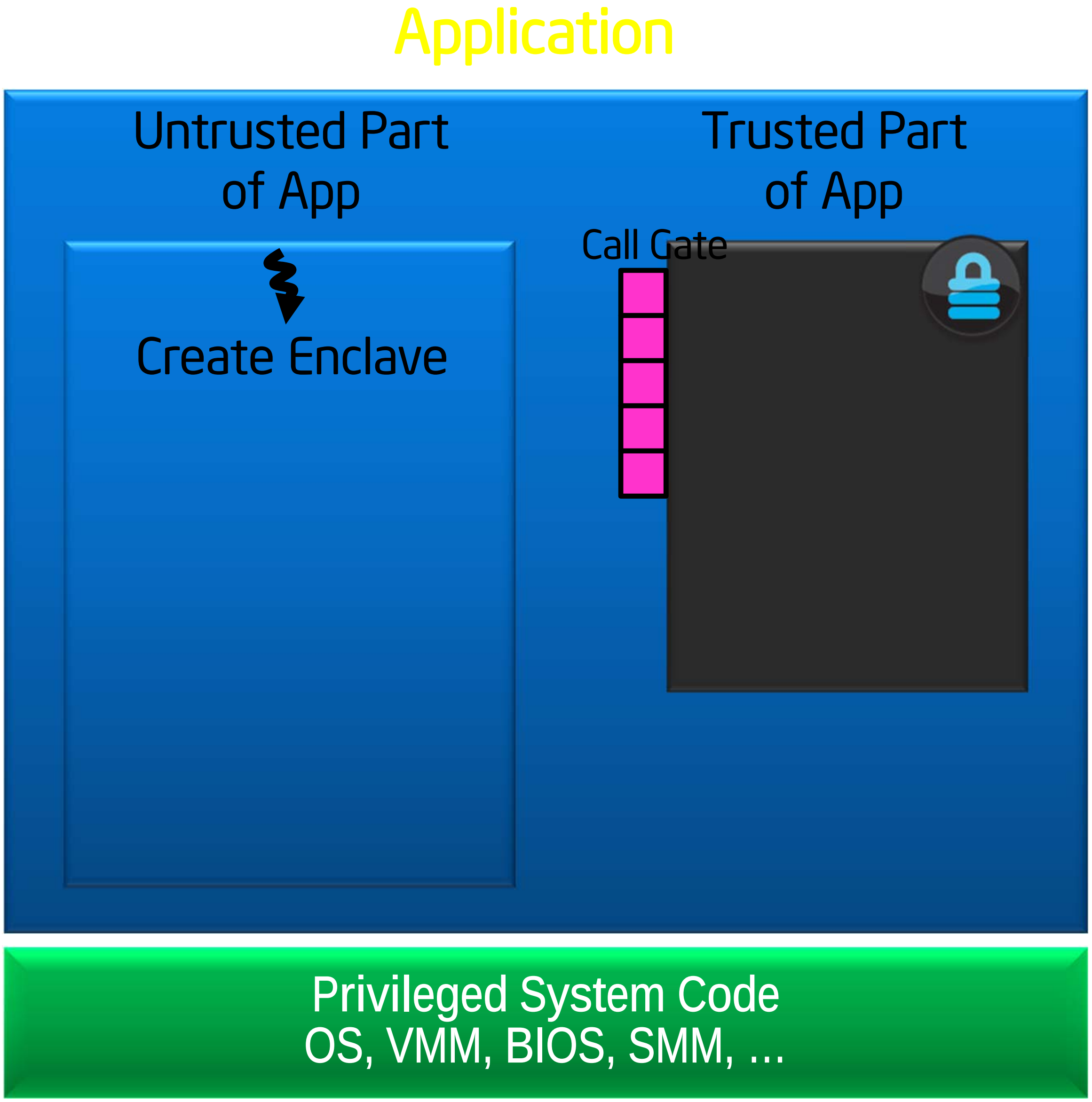
Privileged System Code
OS, VMM, BIOS, SMM, ...

How SE Works: Protection vs. Software Attack



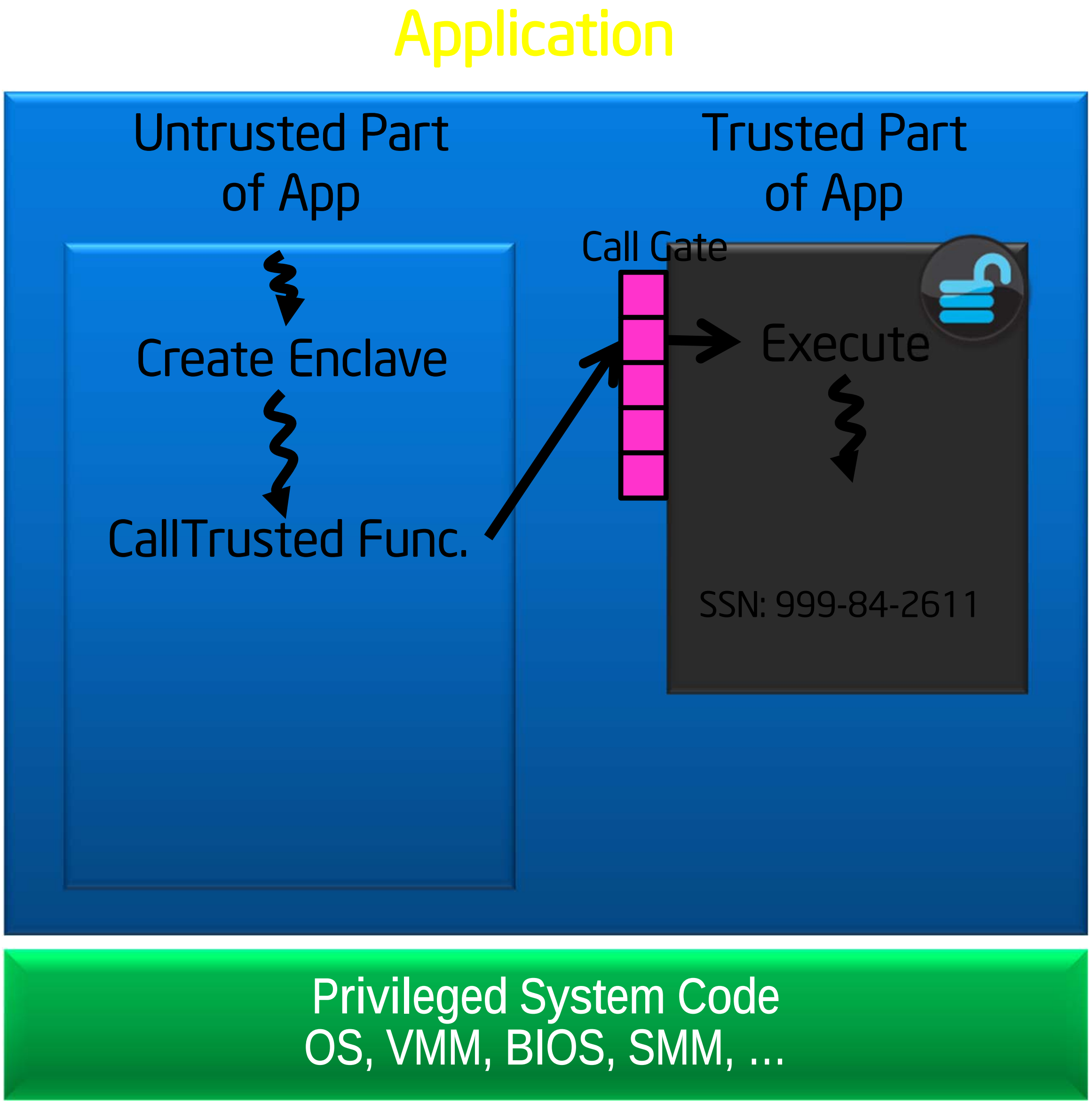
1. App is built with trusted and untrusted parts

How SE Works: Protection vs. Software Attack



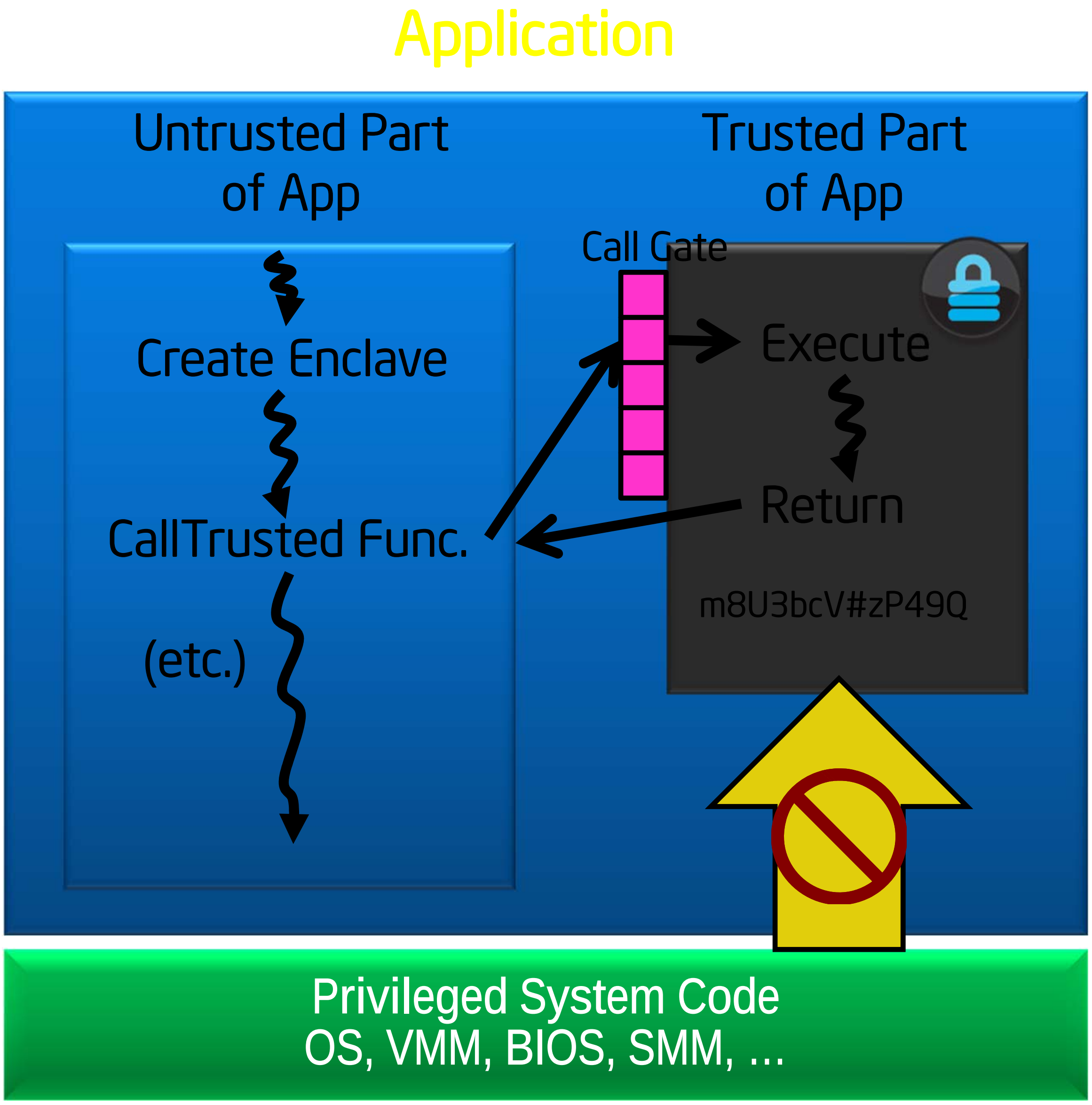
1. App is built with trusted and untrusted parts
2. App runs & creates enclave which is placed in trusted memory

How SE Works: Protection vs. Software Attack



1. App is built with trusted and untrusted parts
2. App runs & creates enclave which is placed in trusted memory
3. Trusted function is called; code running inside enclave sees data in clear; external access to data is denied

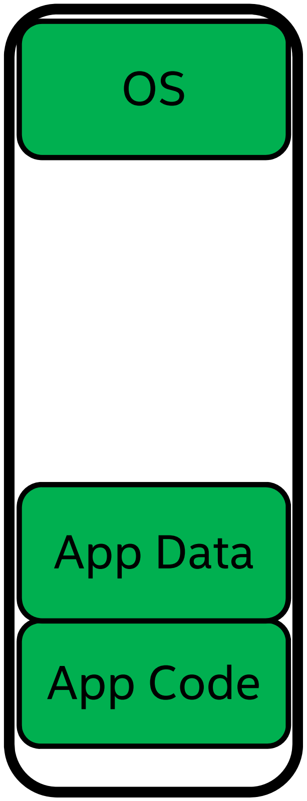
How SE Works: Protection vs. Software Attack



1. App is built with trusted and untrusted parts
2. App runs & creates enclave which is placed in trusted memory
3. Trusted function is called; code running inside enclave sees data in clear; external access to data is denied
4. Function returns; enclave data remains in trusted memory

SGX Programming Environment

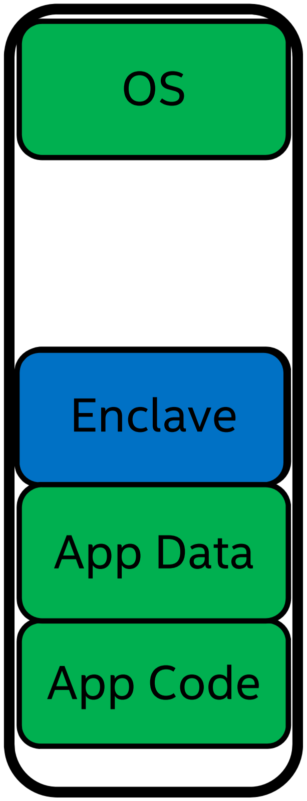
Trusted execution environment embedded in a process



User Process

SGX Programming Environment

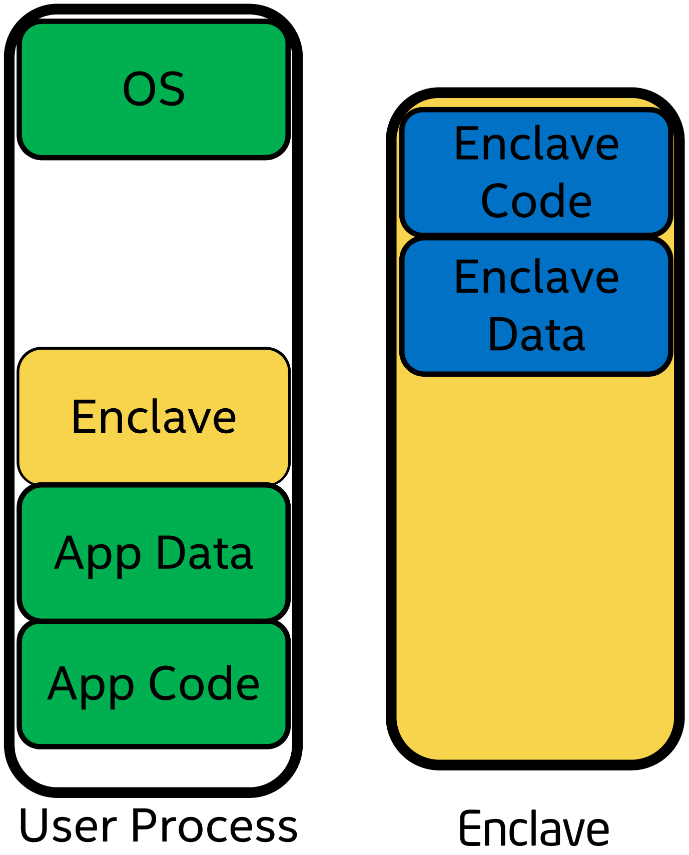
Trusted execution environment embedded in a process



User Process

SGX Programming Environment

Trusted execution environment embedded in a process



With its own code and data

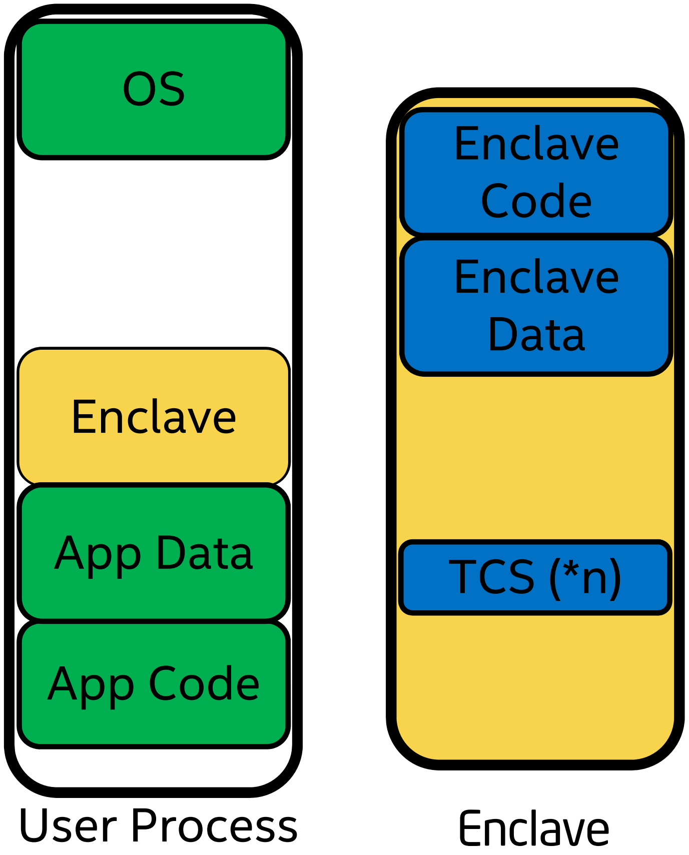
Provide Confidentiality

Provide integrity

With controlled entry points

SGX Programming Environment

Trusted execution environment embedded in a process



With its own code and data

Provide Confidentiality

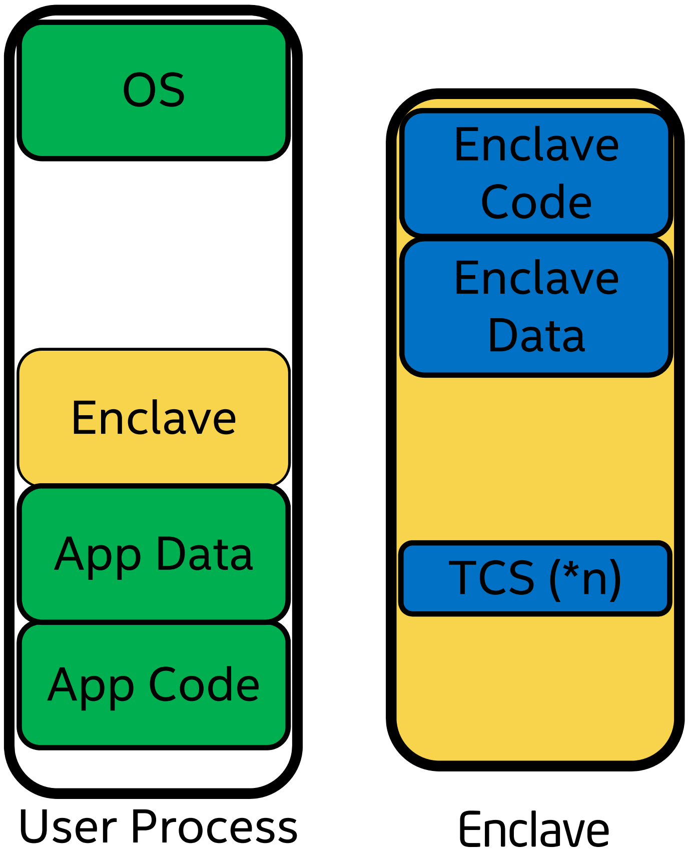
Provide integrity

With controlled entry points

Supporting multiple threads

SGX Programming Environment

Trusted execution environment embedded in a process



With its own code and data

Provide Confidentiality

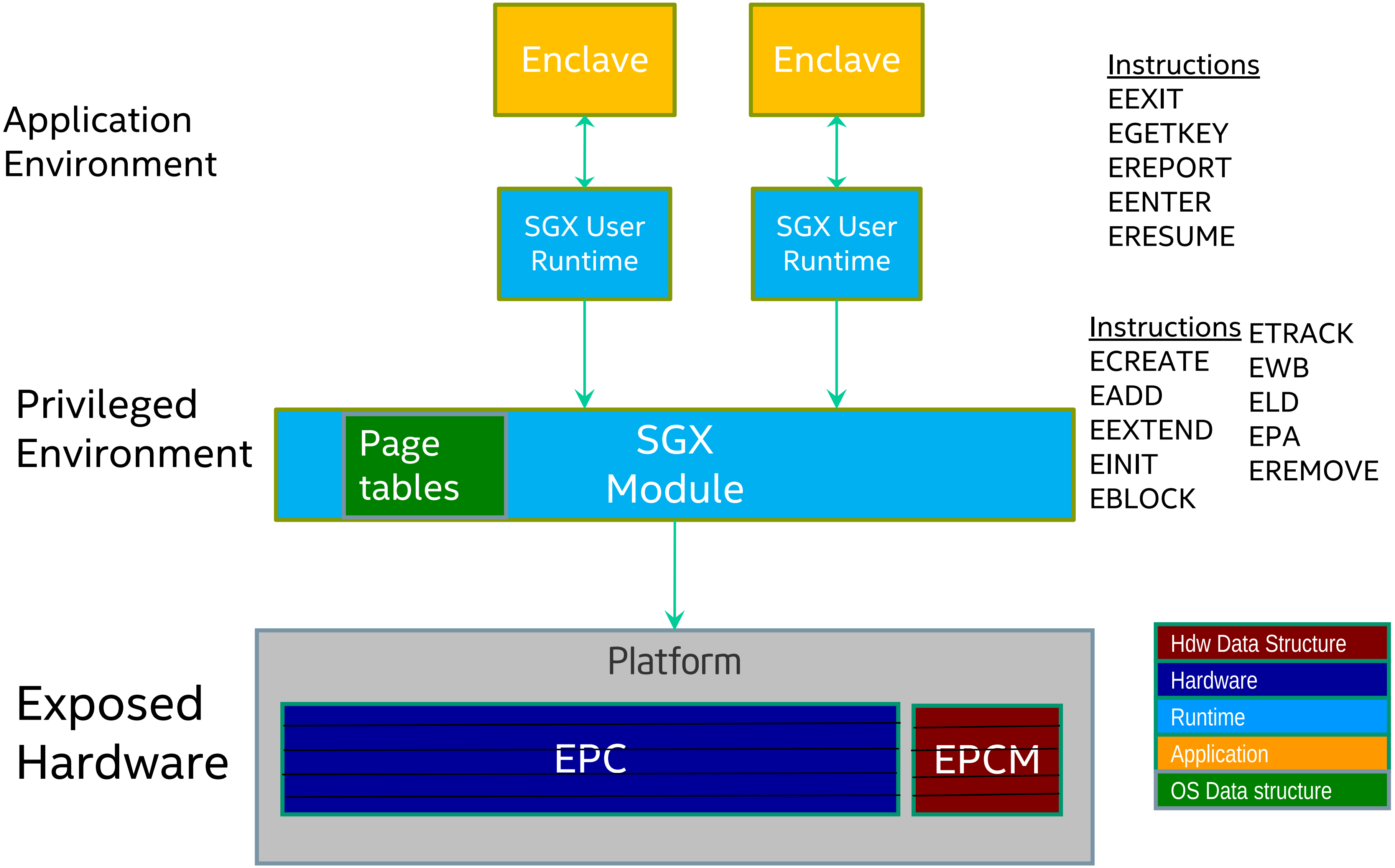
Provide integrity

With controlled entry points

Supporting multiple threads

With full access to app memory

SGX High-level HW/SW Picture



Application Environment

Enclave

Enclave

SGX User Runtime

SGX User Runtime

Privileged Environment

Page tables

SGX Module

Exposed Hardware

Platform

EPC

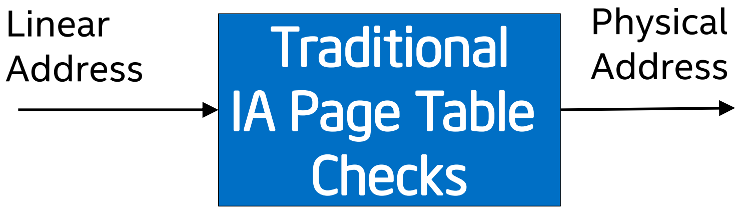
EPCM

Instructions
EEXIT
EGETKEY
EREPORT
EENTER
ERESUME

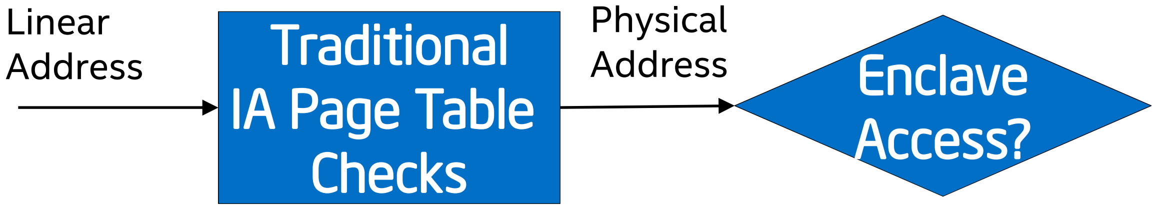
Instructions
ECREATE
EADD
EEXTEND
EINIT
EBLOCK
ETRACK
EWB
ELD
EPA
EREMOVE

🟥 Hdw Data Structure
🟦 Hardware
🟦 Runtime
🟧 Application
🟩 OS Data structure

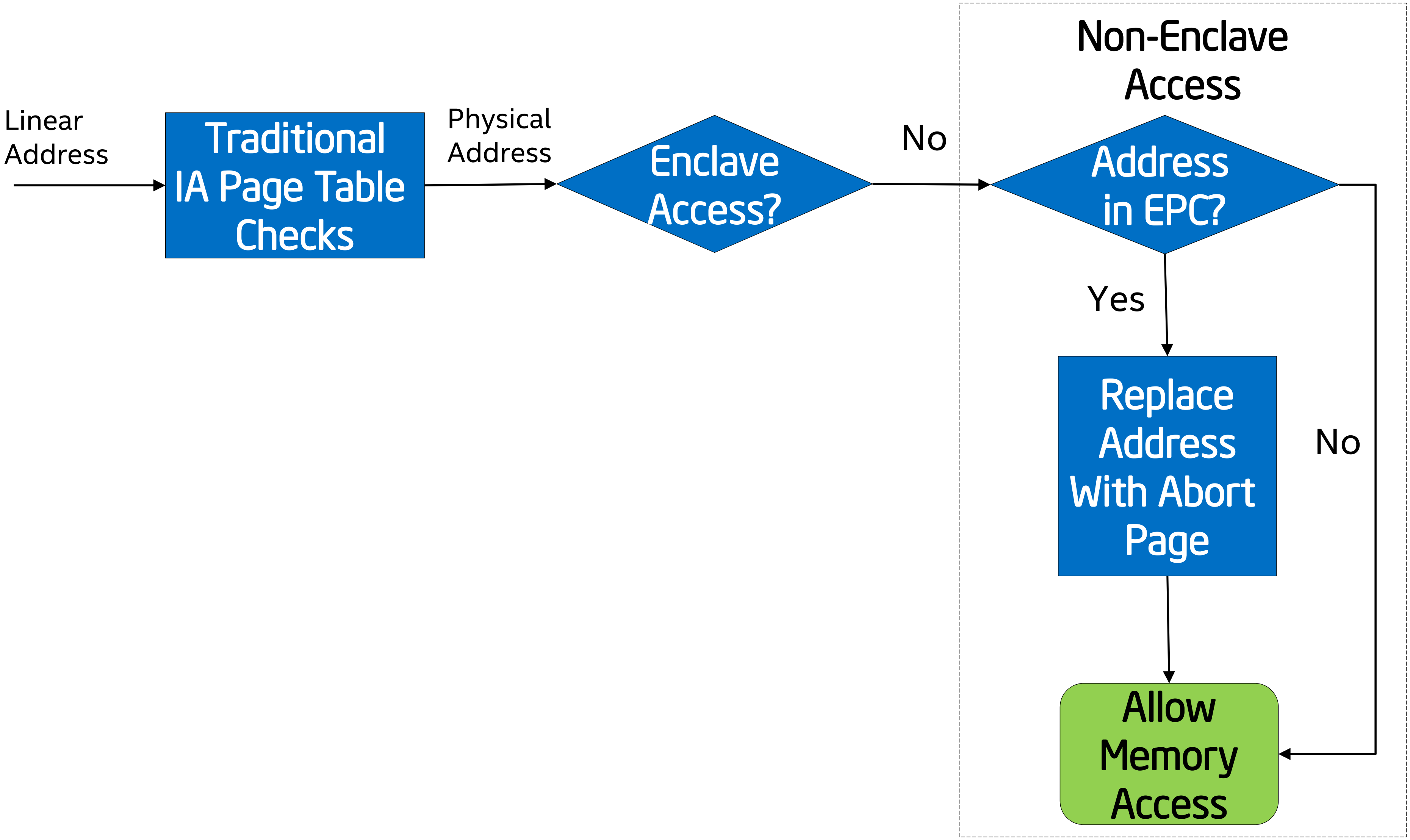
SGX Access Control



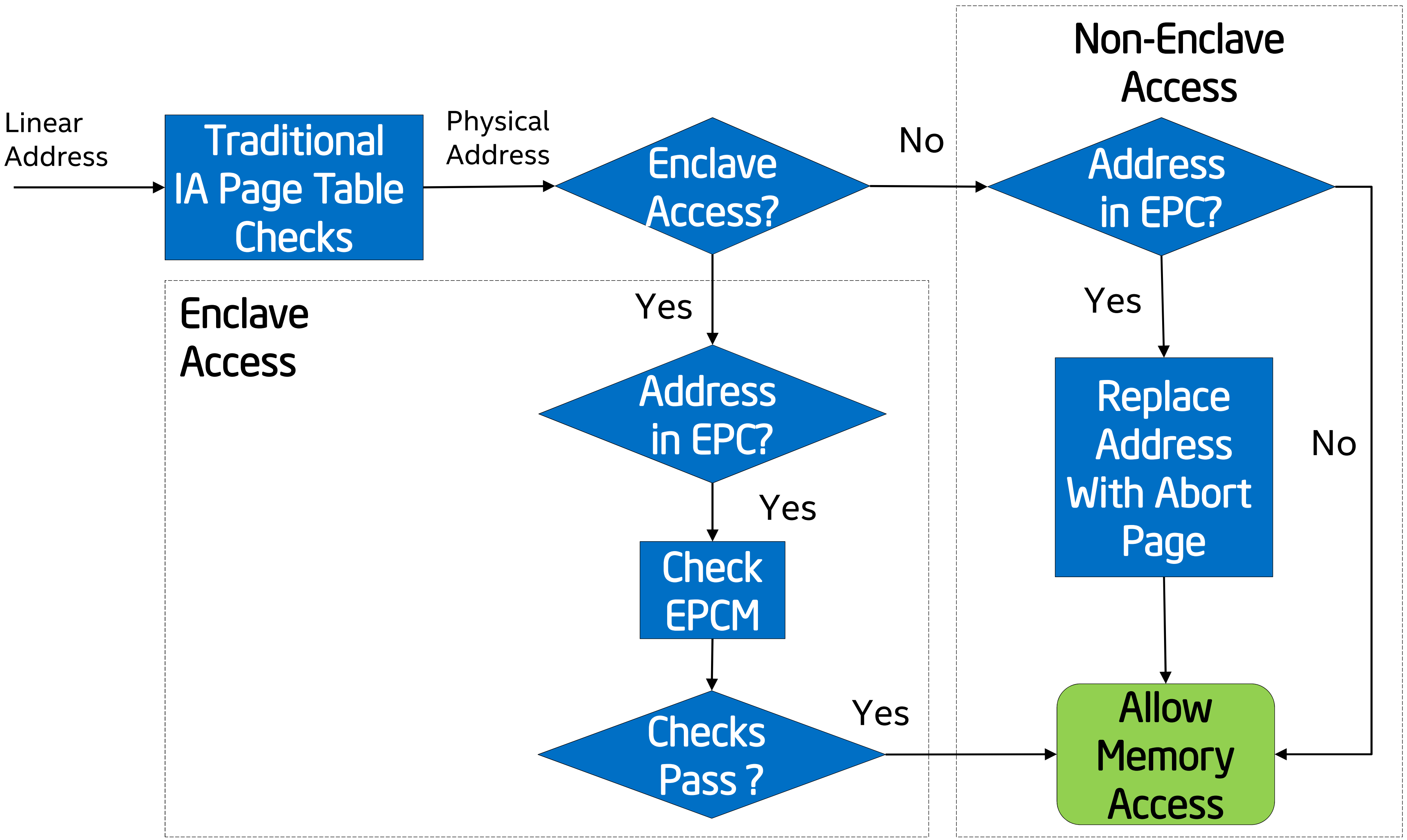
SGX Access Control



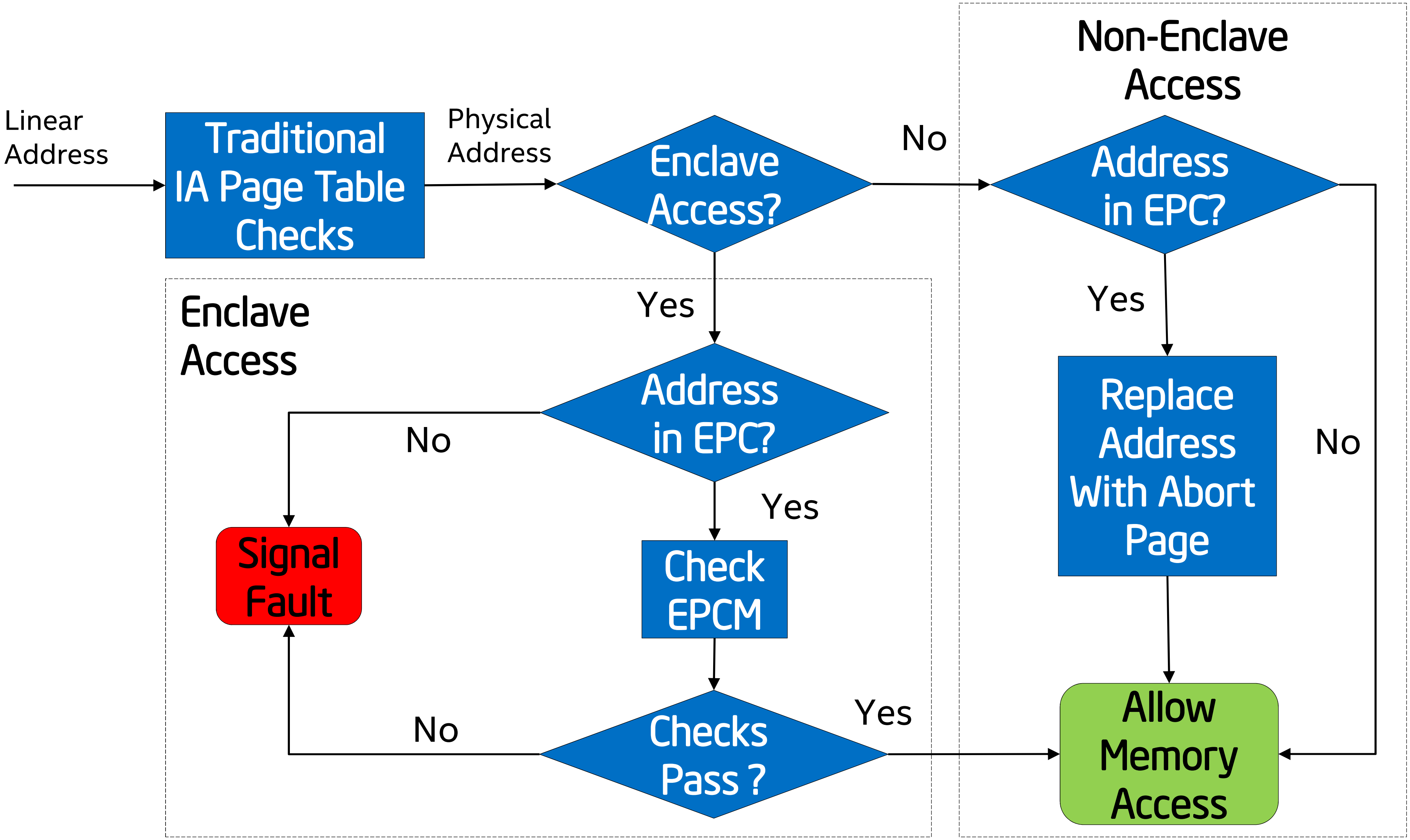
SGX Access Control



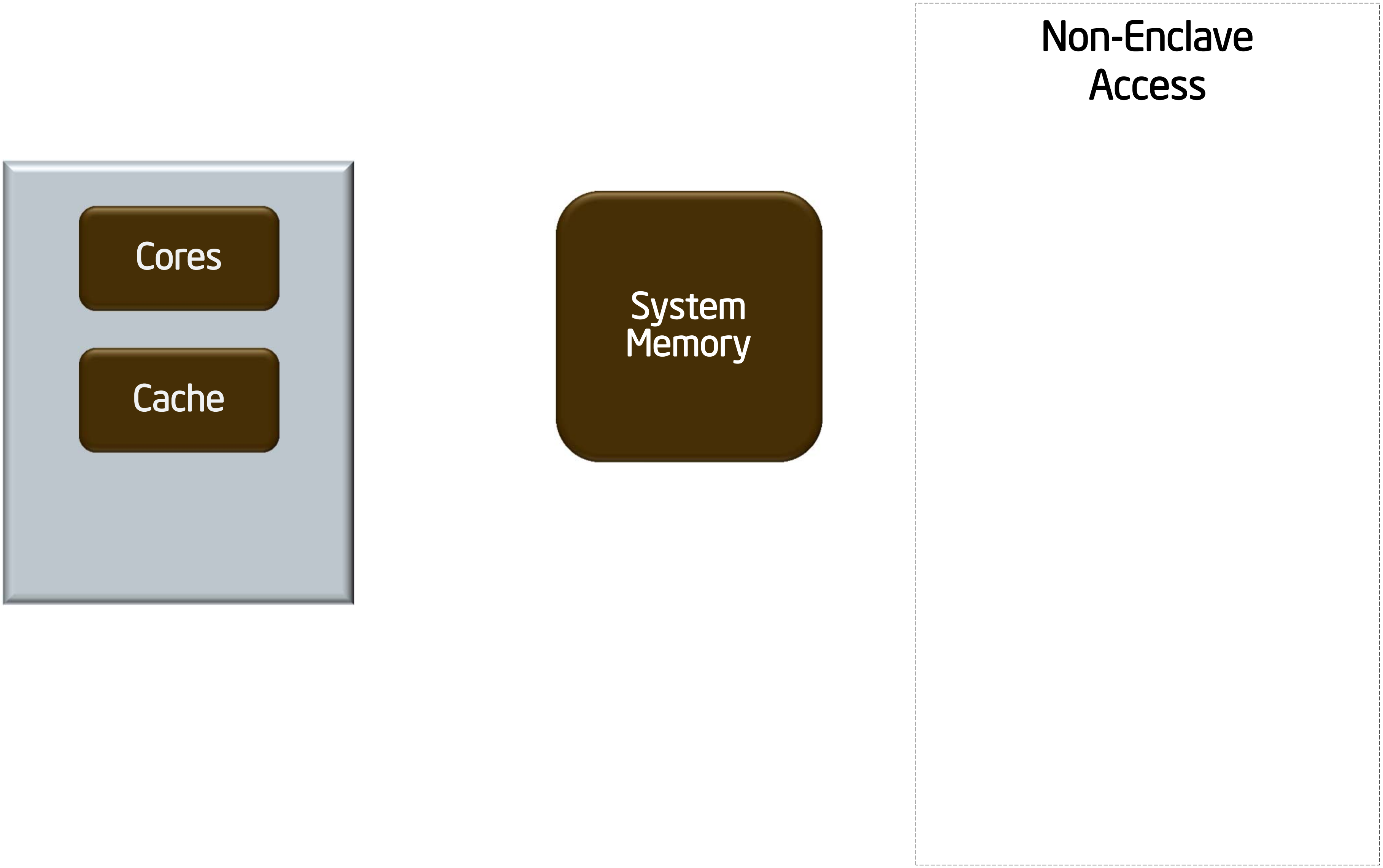
SGX Access Control



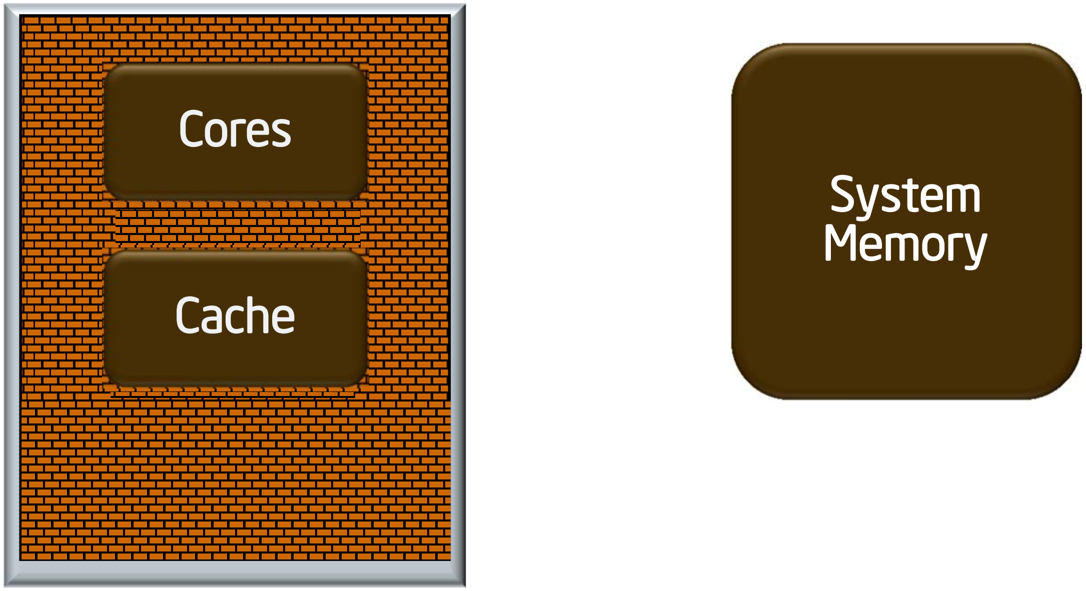
SGX Access Control



Protection vs. Memory Snooping Attacks



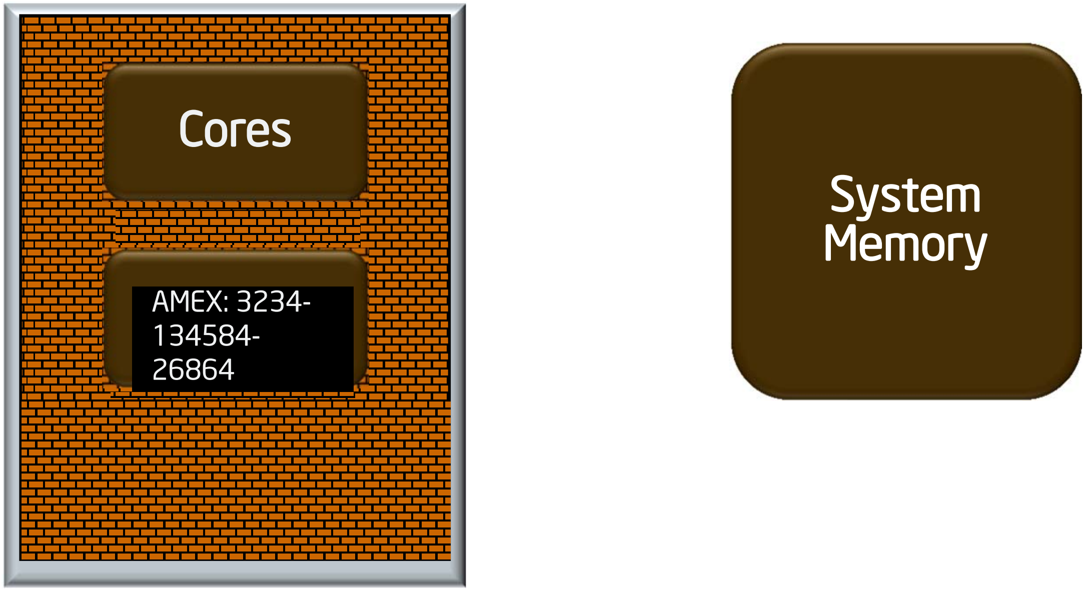
Protection vs. Memory Snooping Attacks



Non-Enclave Access

- Security perimeter is the CPU package boundary

Protection vs. Memory Snooping Attacks



Cores

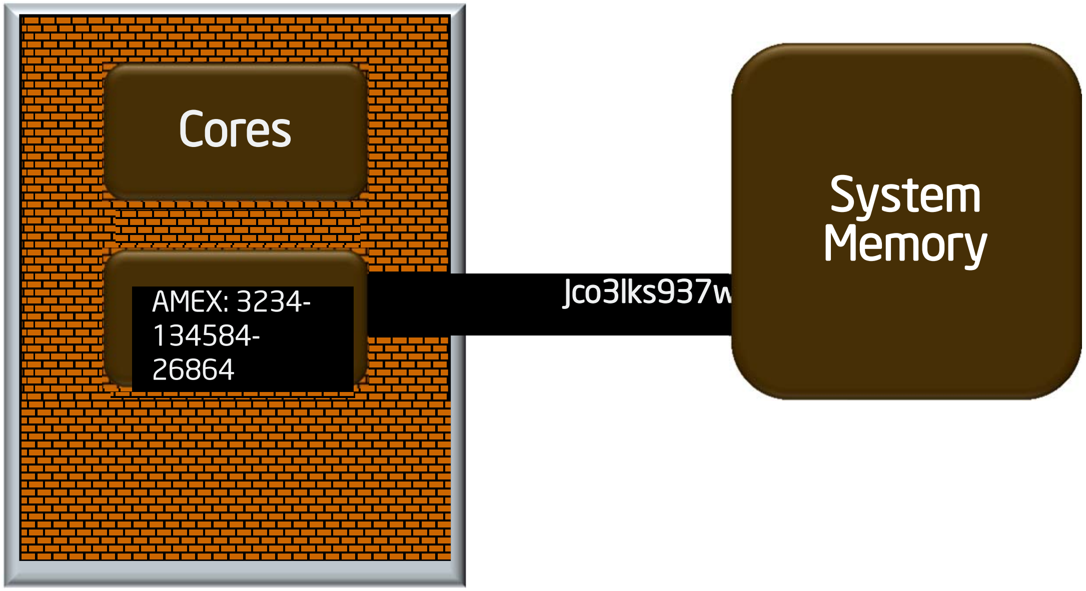
AMEX: 3234-134584-26864

System Memory

Non-Enclave Access

- Security perimeter is the CPU package boundary
- Data and code unencrypted inside CPU package

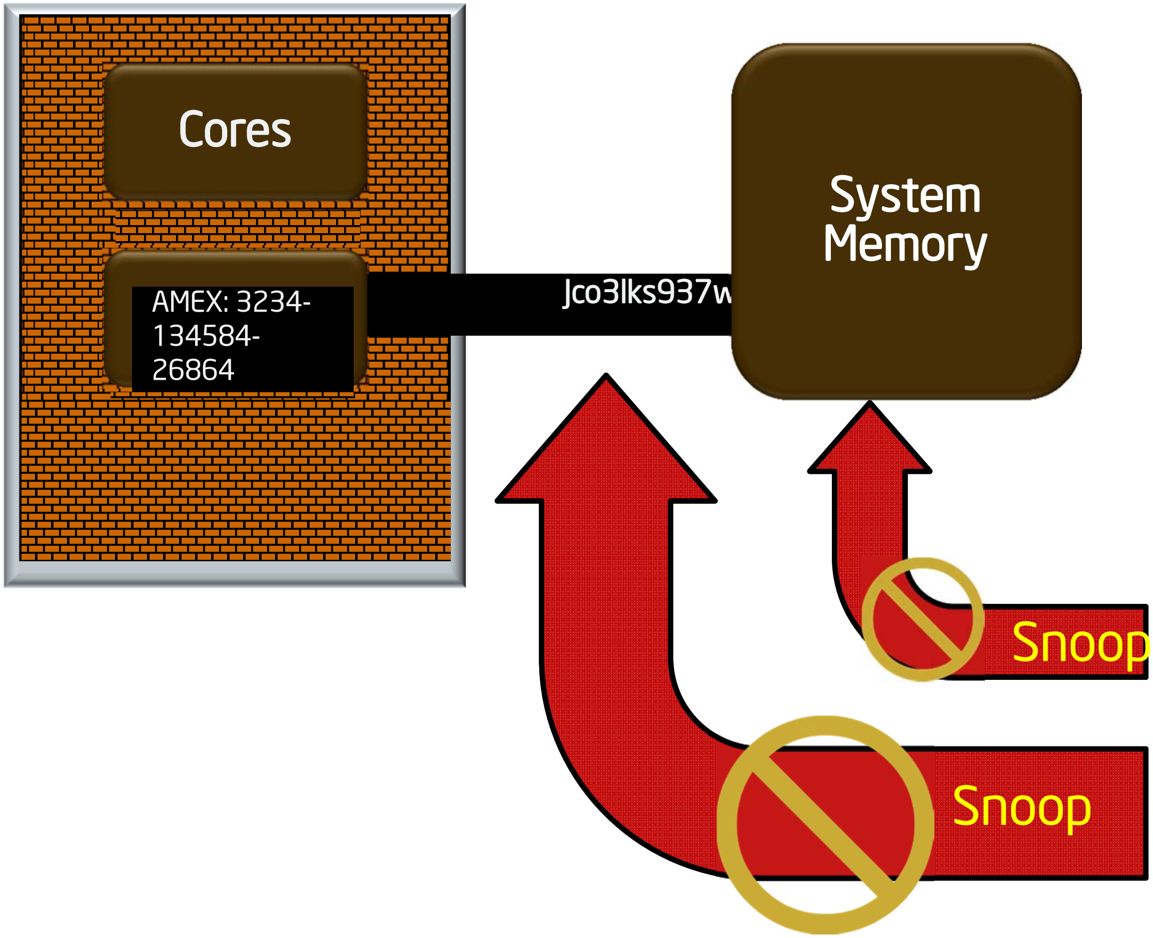
Protection vs. Memory Snooping Attacks



Non-Enclave Access

- Security perimeter is the CPU package boundary
- Data and code unencrypted inside CPU package
- Data and code outside CPU package is encrypted and/or integrity checked

Protection vs. Memory Snooping Attacks



Non-Enclave Access

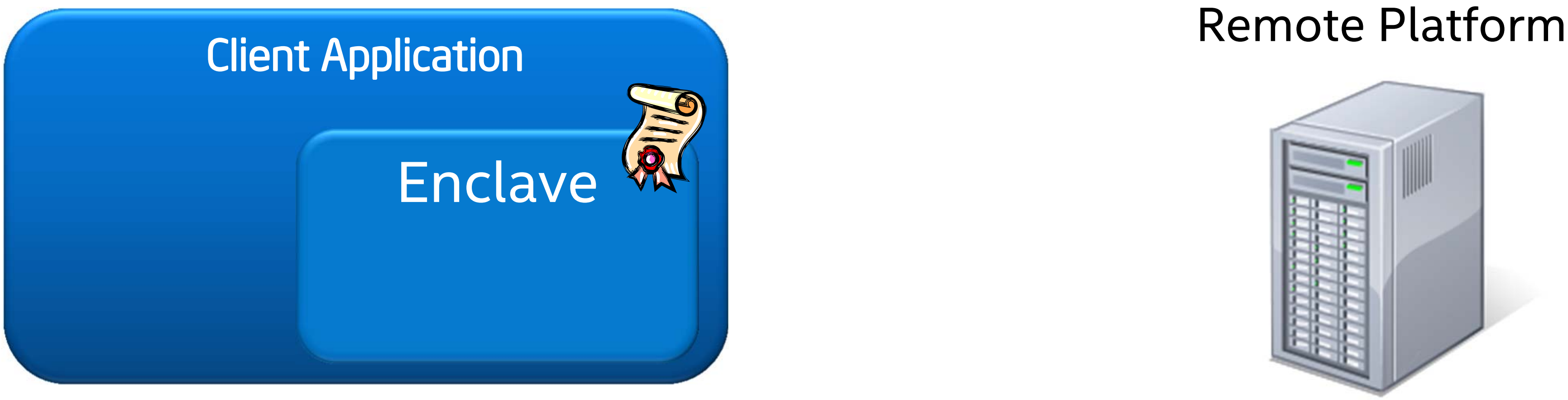
- Security perimeter is the CPU package boundary
- Data and code unencrypted inside CPU package
- Data and code outside CPU package is encrypted and/or integrity checked
- External memory reads and bus snoops see only encrypted data

Critical Feature: Attestation and Sealing

Client Application

Remote Platform

Critical Feature: Attestation and Sealing



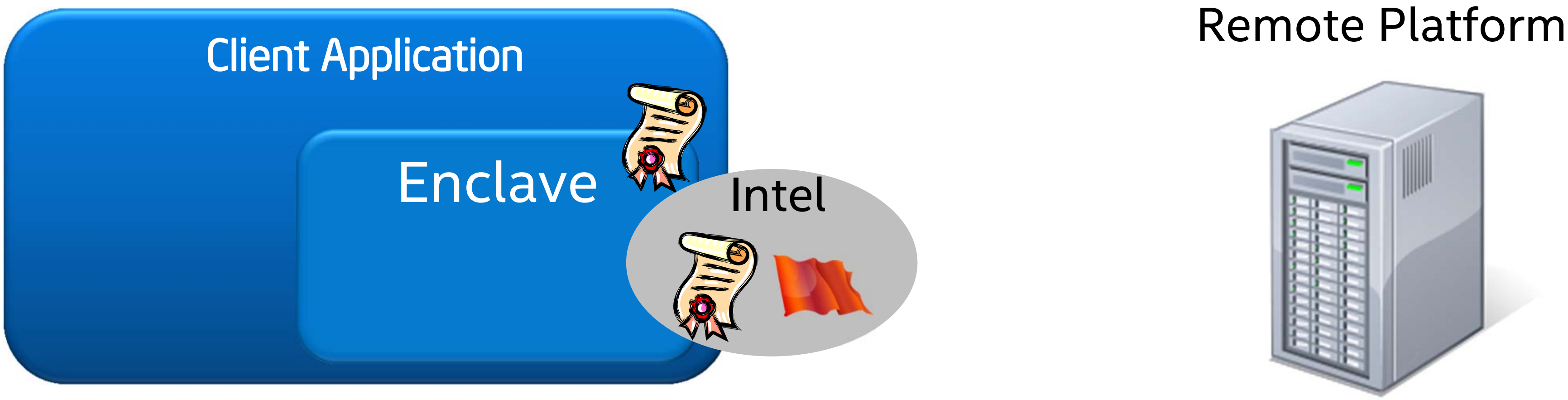
Client Application

Enclave

Remote Platform

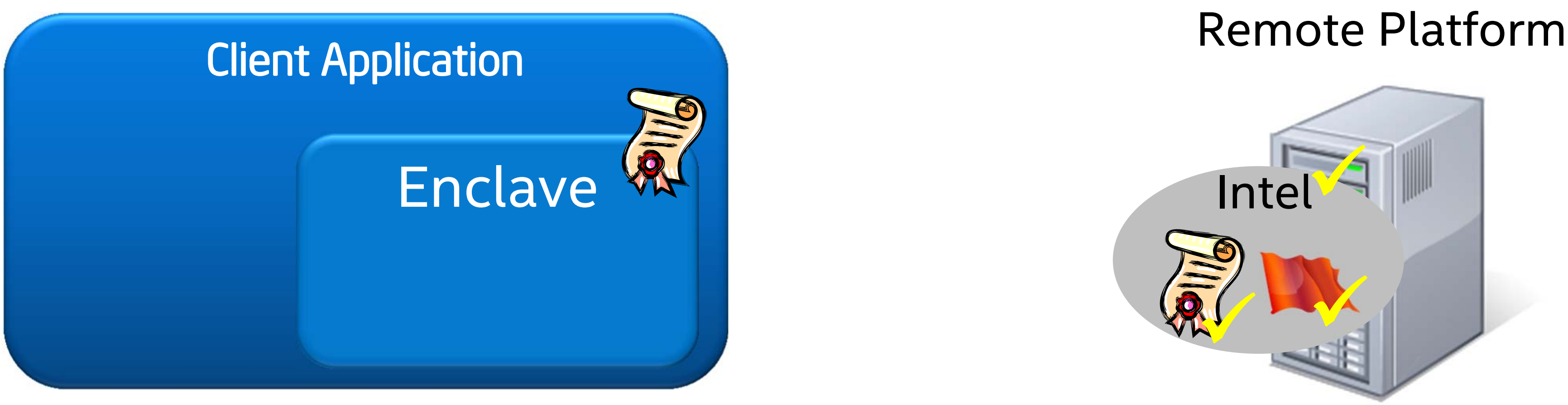
1. Enclave built & measured

Critical Feature: Attestation and Sealing



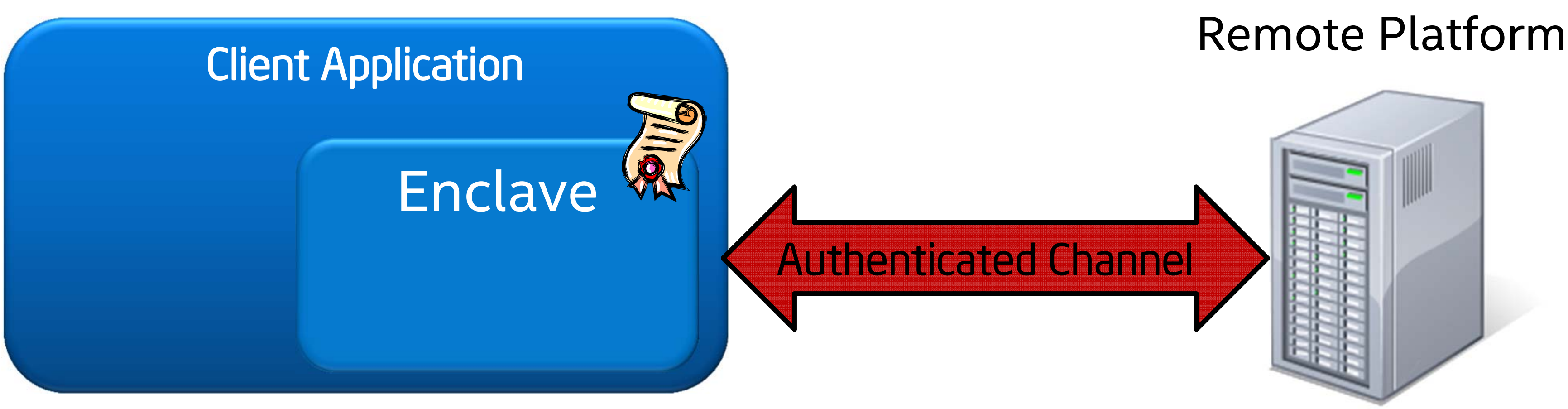
1. Enclave built & measured
2. Enclave requests REPORT (HW-signed blob that includes enclave identity information)

Critical Feature: Attestation and Sealing



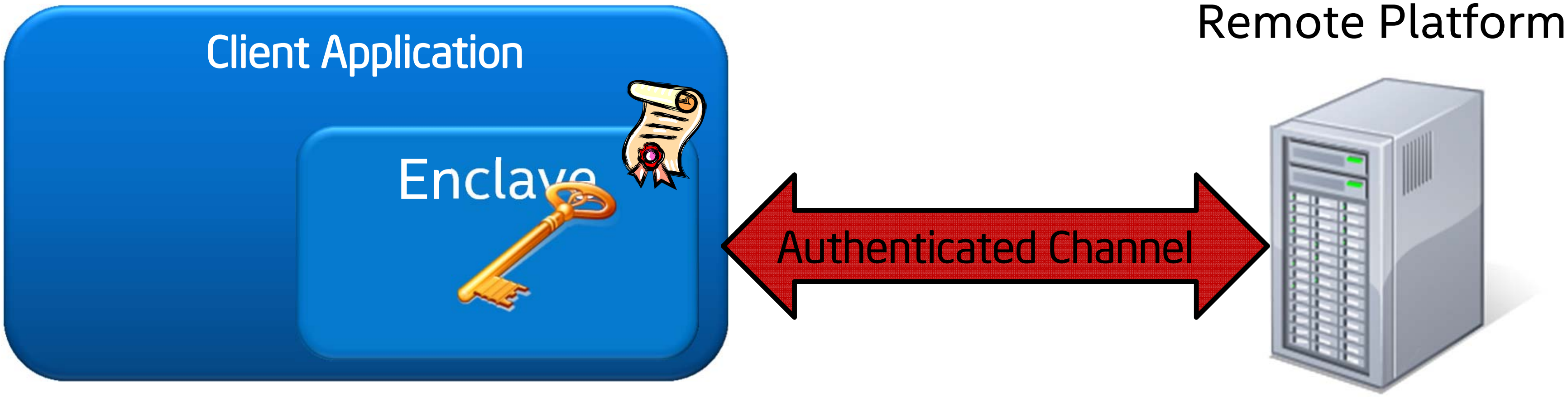
1. Enclave built & measured
2. Enclave requests REPORT (HW-signed blob that includes enclave identity information)
3. REPORT sent to server & verified

Critical Feature: Attestation and Sealing



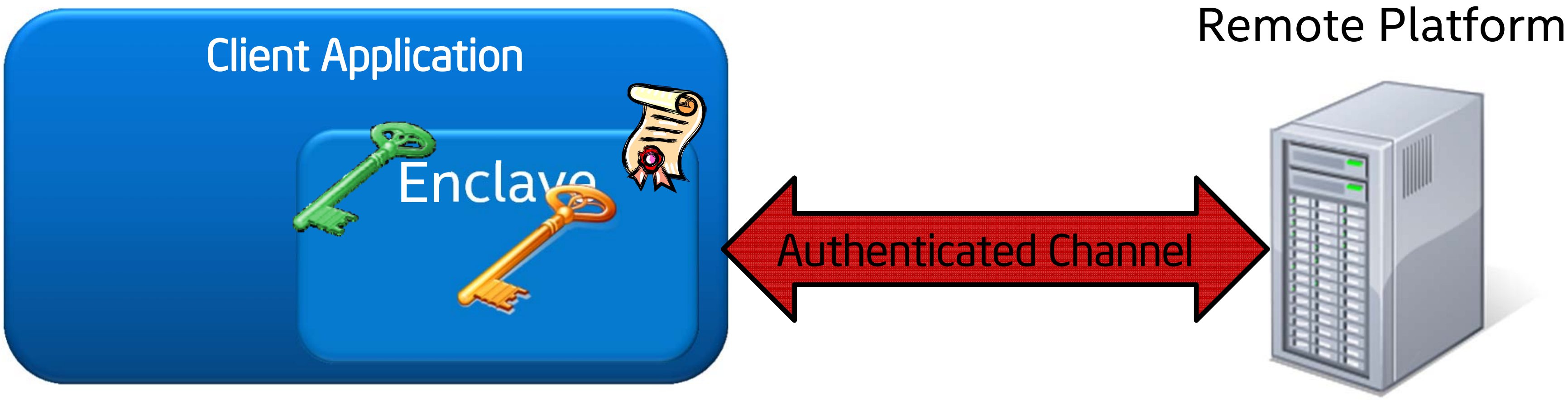
1. Enclave built & measured
2. Enclave requests REPORT (HW-signed blob that includes enclave identity information)
3. REPORT sent to server & verified

Critical Feature: Attestation and Sealing



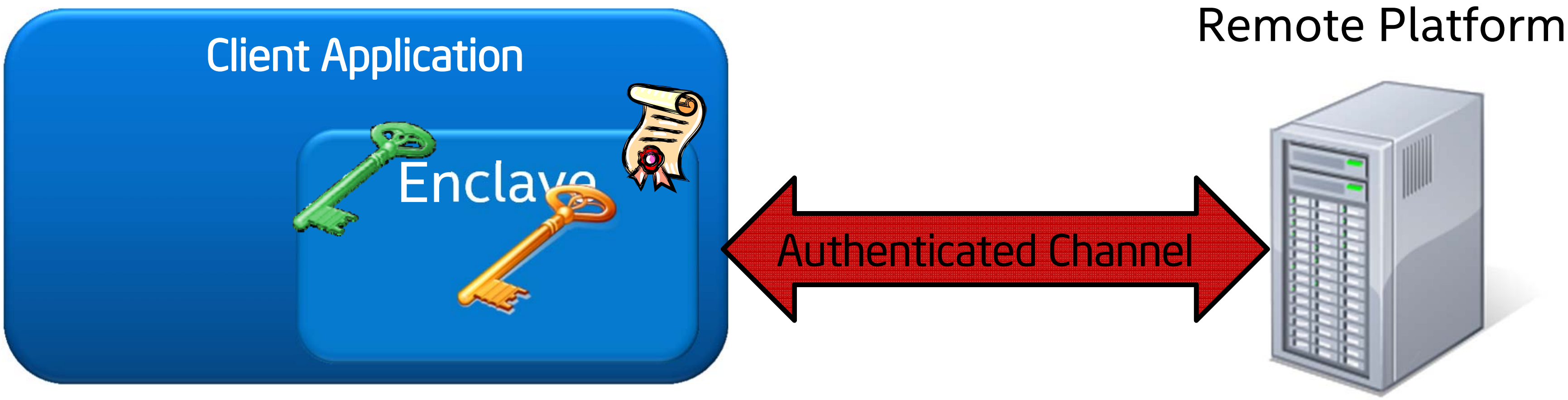
1. Enclave built & measured
2. Enclave requests REPORT (HW-signed blob that includes enclave identity information)
3. REPORT sent to server & verified
4. Application Key sent to enclave, first secret provisioned

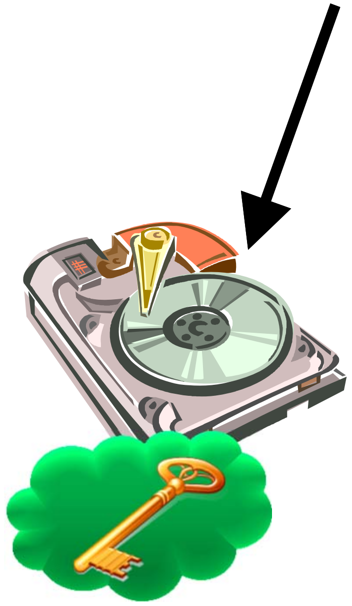
Critical Feature: Attestation and Sealing



1. Enclave built & measured
2. Enclave requests REPORT (HW-signed blob that includes enclave identity information)
3. REPORT sent to server & verified
4. Application Key sent to enclave, first secret provisioned
5. Enclave-platform-specific Sealing Key generated (EGETKEY)

Critical Feature: Attestation and Sealing





1. Enclave built & measured
2. Enclave requests REPORT (HW-signed blob that includes enclave identity information)
3. REPORT sent to server & verified
4. Application Key sent to enclave, first secret provisioned
5. Enclave-platform-specific Sealing Key generated (EGETKEY)
6. Application Key encrypted via Sealing Key & stored for later (offline) use

Security Vision for a Future Platform

End User: Even naïve users can safely manage valuable financial accounts

Corporate IT: Employee can handle corporate documents on unmanaged platforms, and documents are protected from theft even if employee has malware and has not kept up with patches.

Cloud Service: Cloud customers can execute workloads in the cloud with assurance that their workloads are safe from any malware outside of their workload

App Dev: App developers can develop and deploy protected apps just as easily as other apps.

Example Prototype Usages

Trusted browsing

- Naïve users can safely manage valuable financial accounts

Secure document sharing

- Employee can handle corporate documents on unmanaged platforms, and documents are protected from theft even if employee has loaded malware and has not kept up with patches

Secure video conferencing

- Users are assured of confidential conversations in the presence of malware

In Depth Architecture

Agenda

- Creating an Enclave
- Instantiating an enclave
- Handling exceptions
- EPC page swapping

Creating an Enclave - ISV

- Developer writes and compiles the enclave
  - Trusted functions at the enclave and rest outside
  - SGX1.0 will need to allocate all the memory upfront, SGX2.0 can dynamically allocate memory.
- Developer installs the enclave as DEBUG
  - Any enclave can be run as debug
  - Debug OPTIN is controlled per thread, and can be set via EDBGWR.

Debug enclave

- Debug Enclaves defined by setting the debug bit in Attributes field loaded by ECREATE
  - A debug enclave has the same binary as a production enclave, but uses different keys for attestation and sealing
- 2 Instructions allow a debugger to reach into a debug enclave and read and write data
  - EDBGRD
    - Performs an 8 byte read from a location inside a debug enclave.
  - EDBGWR
    - Performs an 8 byte write to a location inside a debug enclave.
- Breakpoints and Last Branch Record are functional

Creating SIGSTRUCT - ISV

- SIGSTRUCT – Enclave Signature Structure
  - Used to measure the enclave and attributes.
- Measuring the enclave content using SHA-256
- Specifying the attributes
- Setting the ISV information
  - Product ID
  - Security version number - ISVSVN
- Signing the App's SIGSTRUCT using the ISV private key with RSA-3072.

Instantiating an enclave

- ECREATE
  - Creates a unique instance of an enclave, establishes the linear address range, and serves as the enclave's root of trust
    - Enclave mode of operation (32/64)
    - Processor features that enclave supports
    - Debug is allowed or not
  - This information stored within an Secure Enclaves Control Structure (SECS) generated by ECREATE.
- EADD
  - Add Regular (REG) or Thread Control Structure (TCS) pages into the enclave
    - System software responsible for selecting free EPC page, type, and attributes, content of the page and the enclave to which the page added to.
  - Initial EPCM entry to indicate type of page (REG, TCS)
    - Linear address, RWX, associate the page to enclave SECS

Instantiating an enclave – Cont.

- EEXTEND
  - Generates a cryptographic hash of the content of the enclave in 256Byte chunks
    - EEXTEND 16 times for measuring a 4K page
- EINIT
  - Verifies the enclave’s content against the ISV’s signed SIGSTRUCT and initializes the enclave – Mark it ready to be used
    - Validate SIGSTRUCT is signed using SIGSTRUCT public key
    - Enclave measurement matches the measurement specified in SIGSTRUCT.
    - Enclave attributes compatible with SIGSTRUCT
    - Record sealing identity (sealing authority, product id, SVN) in the SECS
- EREMOVE
  - Removes a 4KByte page permanently from the enclave

Enclave flow control - EENTER

- Check that TCS is not busy and flush TLB entries for enclave addresses.
- Transfer control from outside enclave to pre-determined location inside the enclave
- Change the mode of operation to be in enclave mode
- Save RSP/RBP for later restore on enclave asynchronous exit
- Save XCR0 and replace it with enclave XFRM value
- If debug enclave and software wishes to debug the allows traps, breakpoints, single step,
    - Else, set HW so the enclave appears as a single instruction.

Enclave flow control – EEXIT

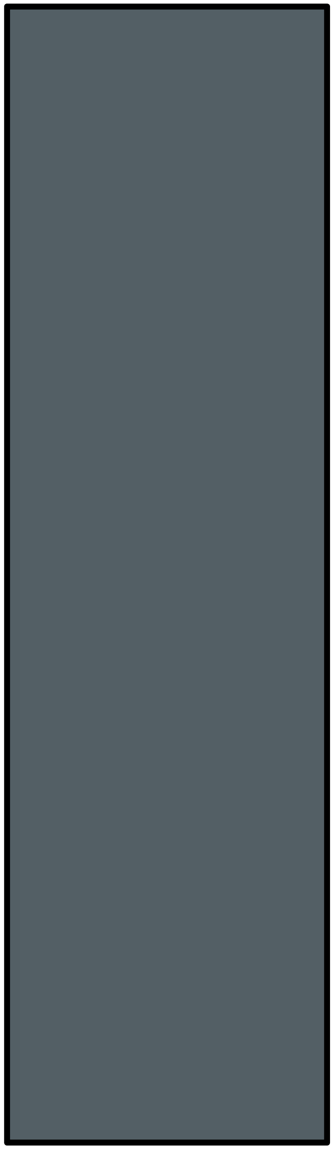
- Clear enclave mode and TLB entries for enclave addresses.
- Transfer control from inside enclave to a location outside specified by RBX
  - Mark TCS as not busy
- Responsibility to clear register state is on enclave writer (run time system)

Life Cycle of An Enclave

Build

1/15

Physical Address Space

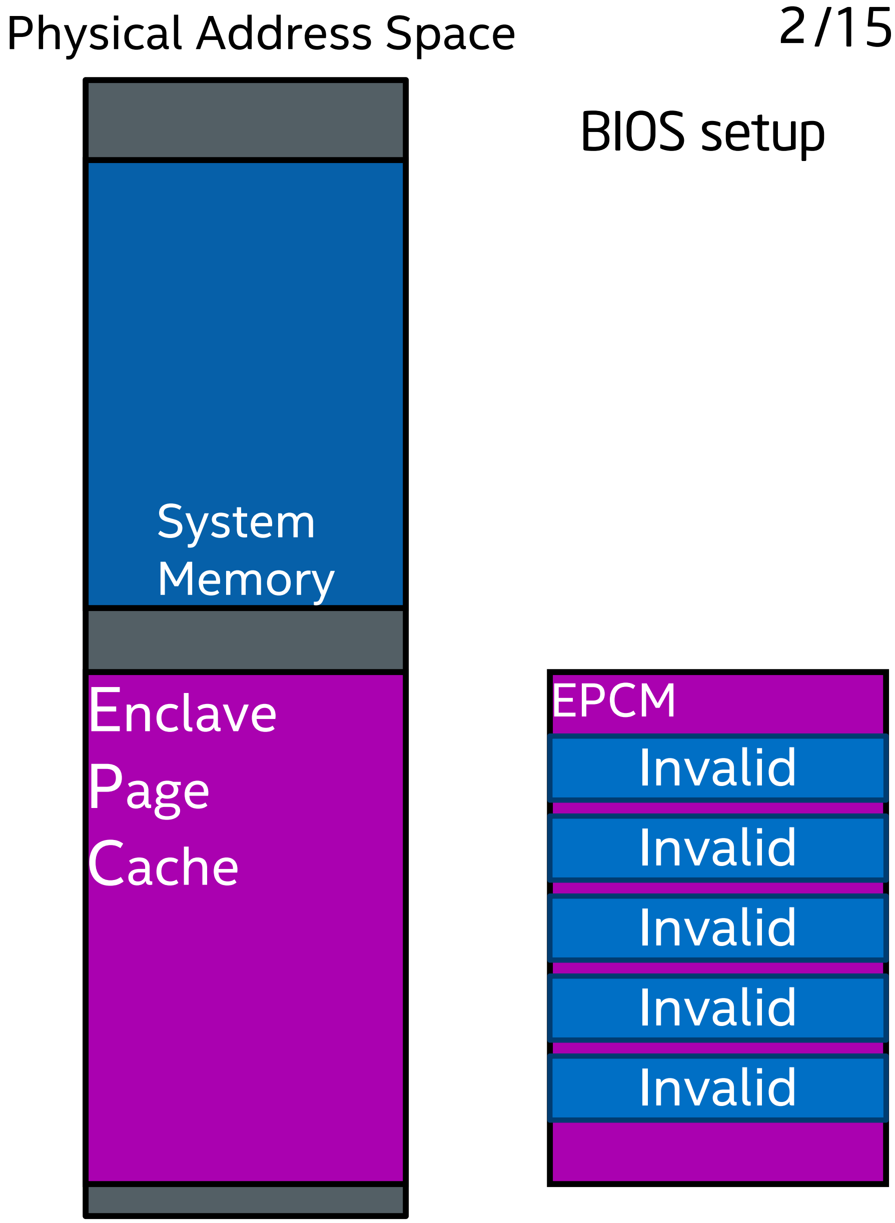


Life Cycle of An Enclave

Build

2/15

BIOS setup

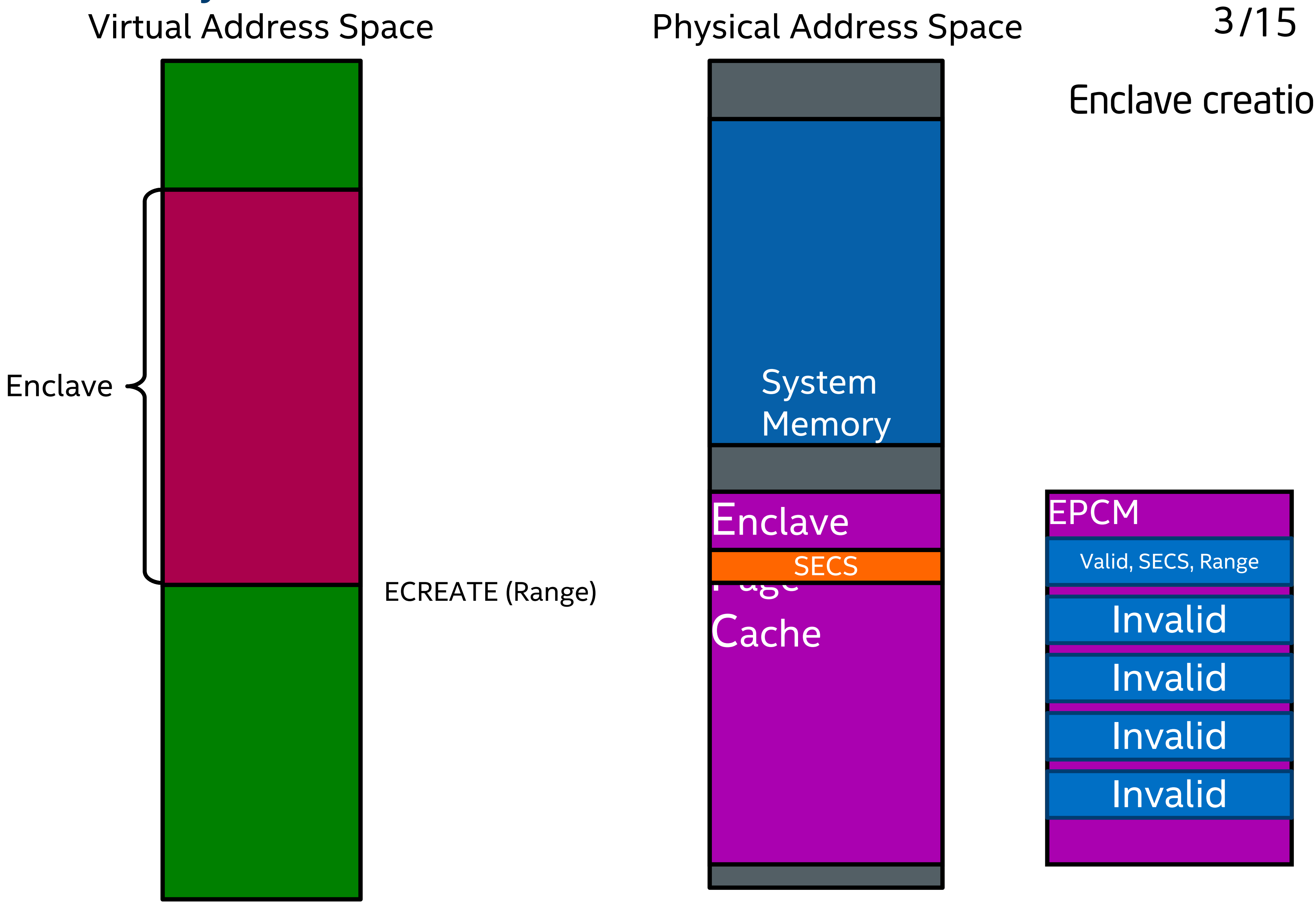


Life Cycle of An Enclave

Build

3/15

Enclave creation

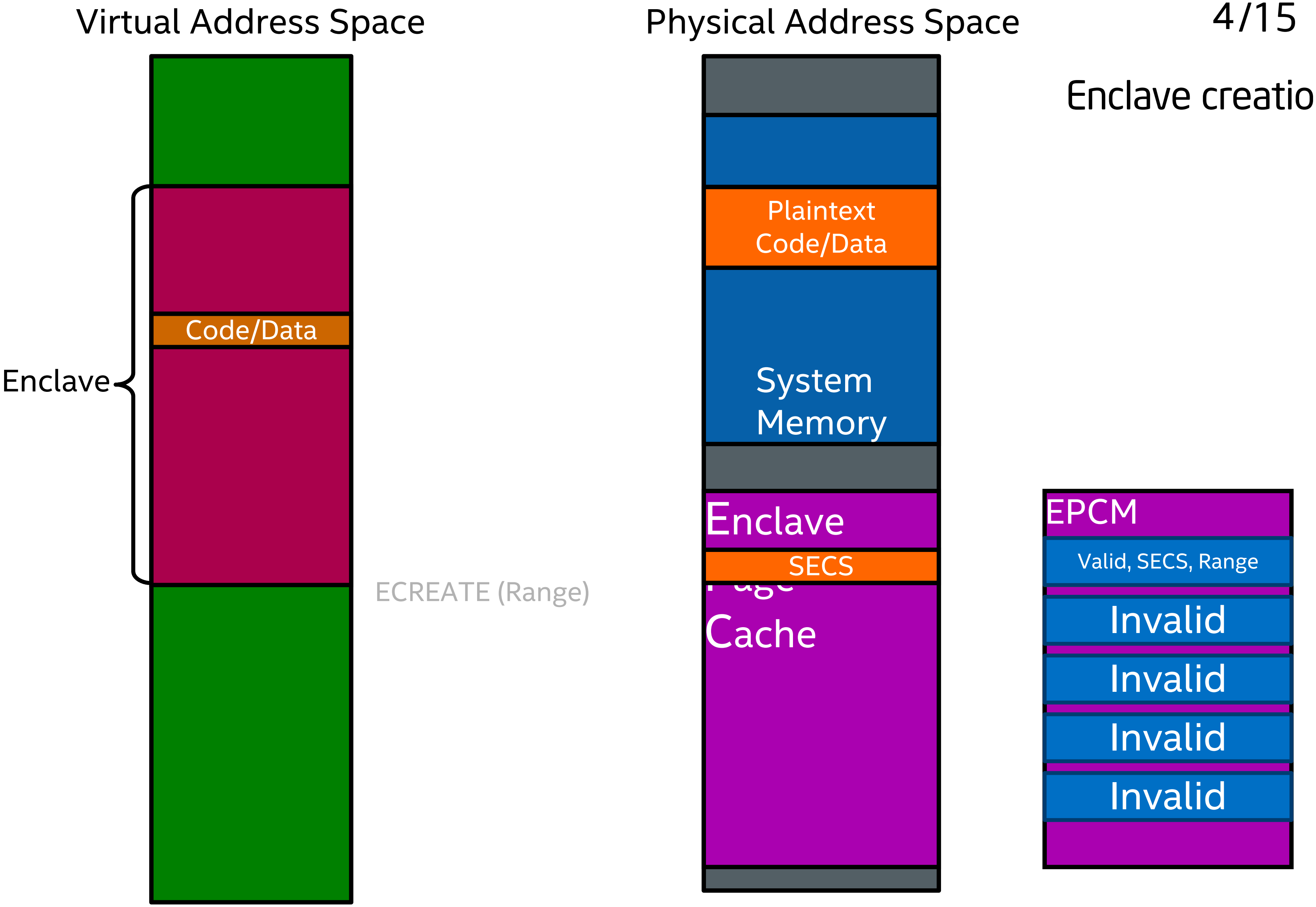


Build

Life Cycle of An Enclave

4/15

Enclave creation

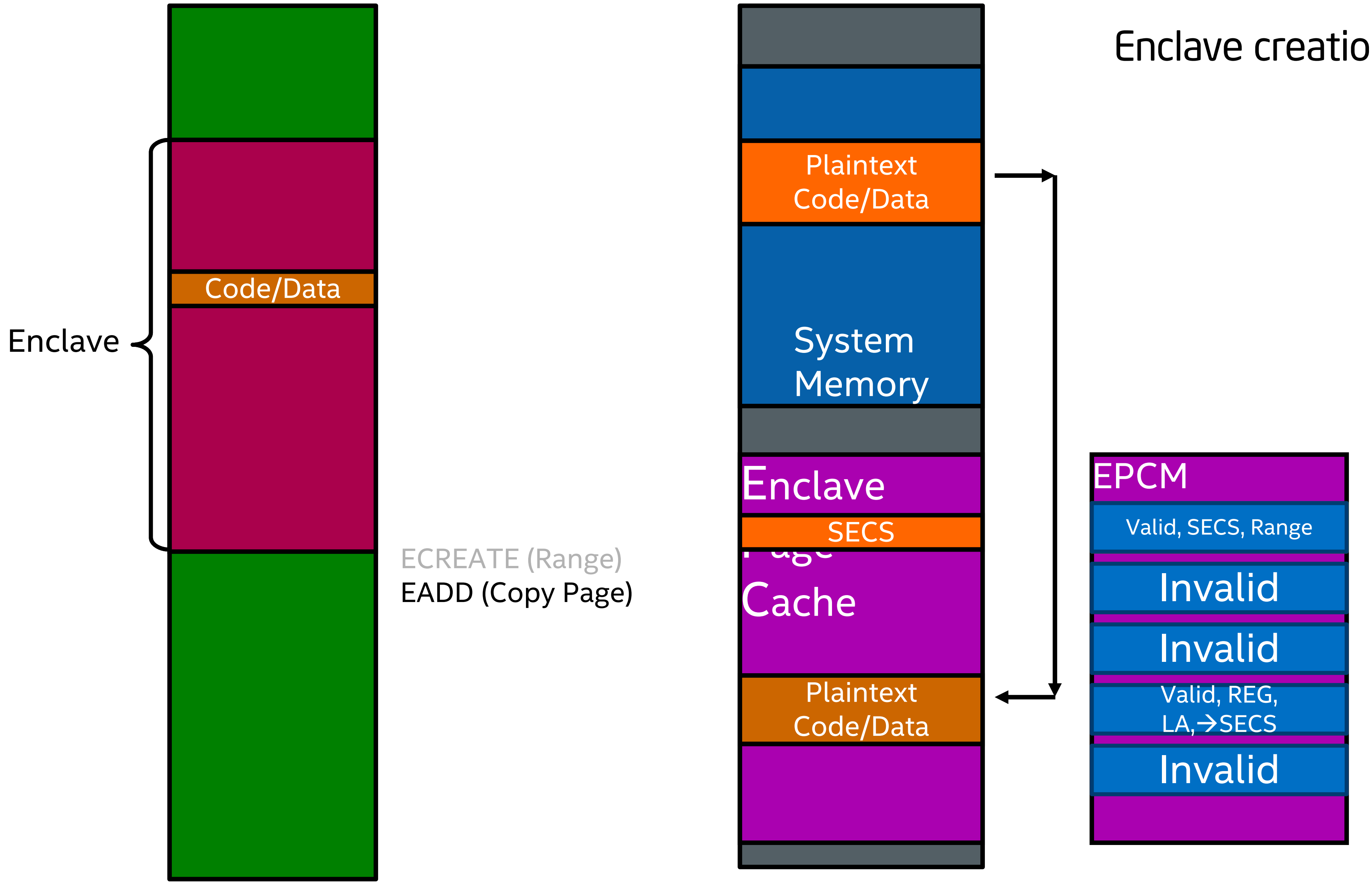


Build

Life Cycle of An Enclave

5/15

Enclave creation



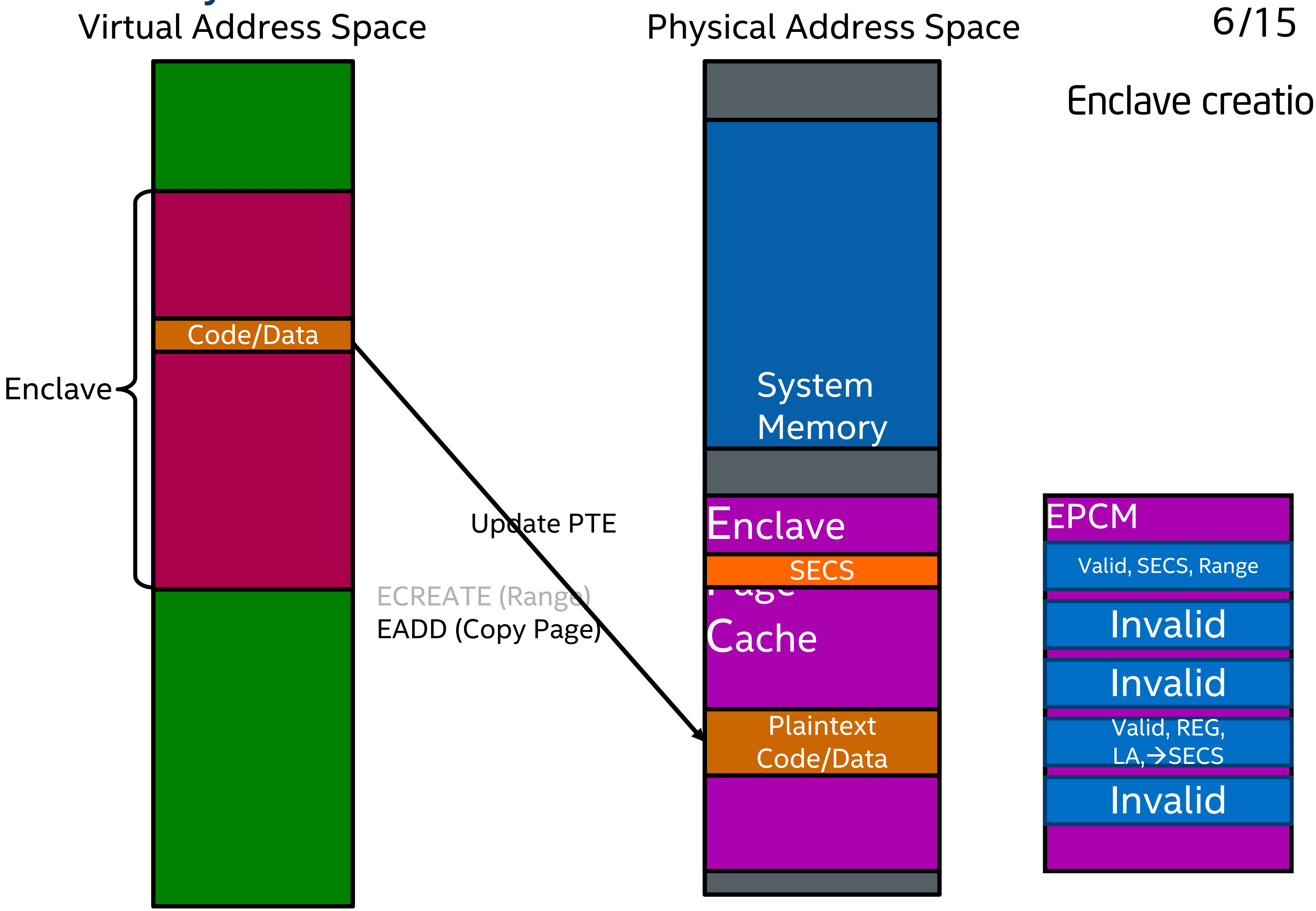
ECREATE (Range)
EADD (Copy Page)

Build

Life Cycle of An Enclave

6/15

Enclave creation

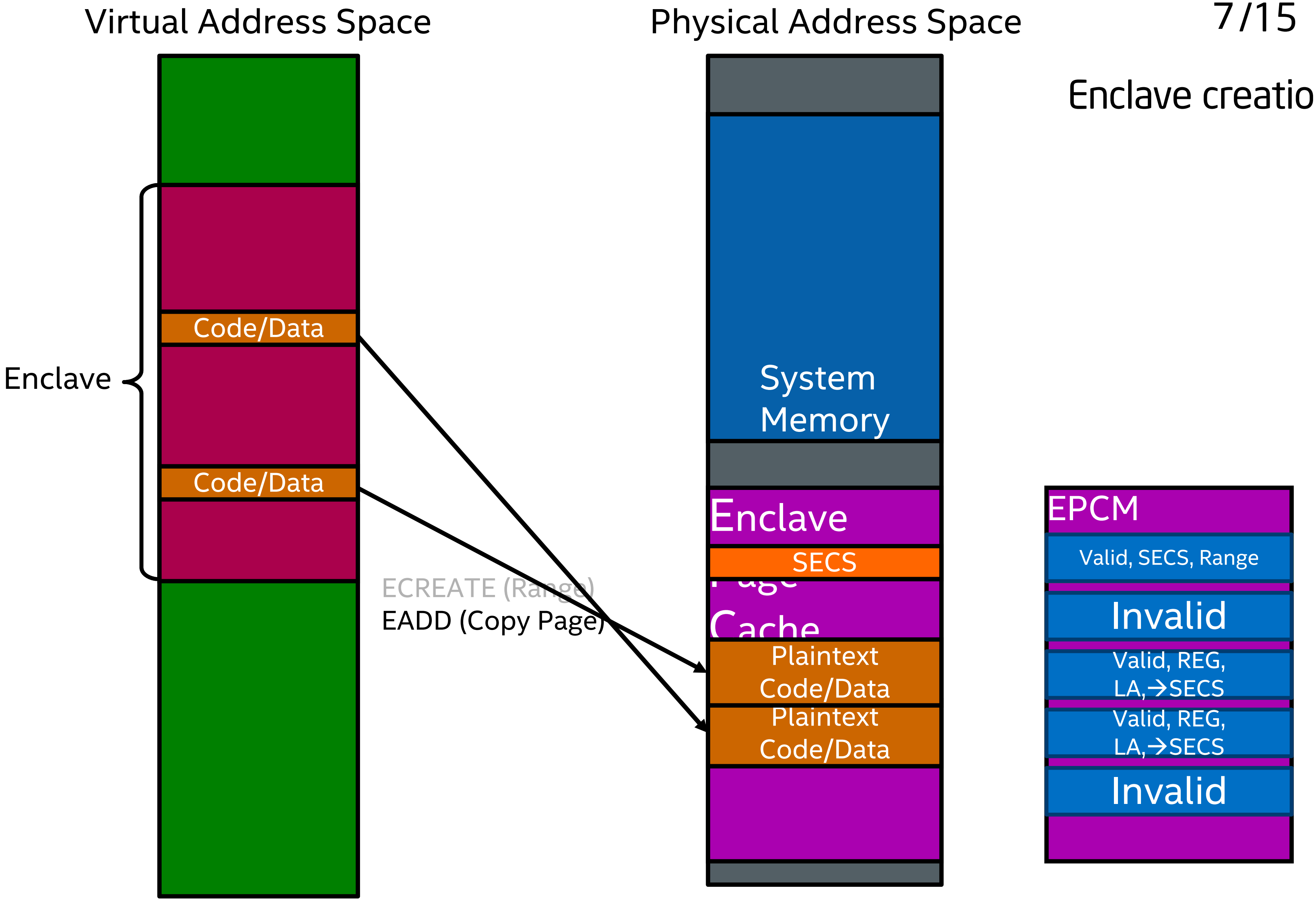


Life Cycle of An Enclave

Build

7/15

Enclave creation



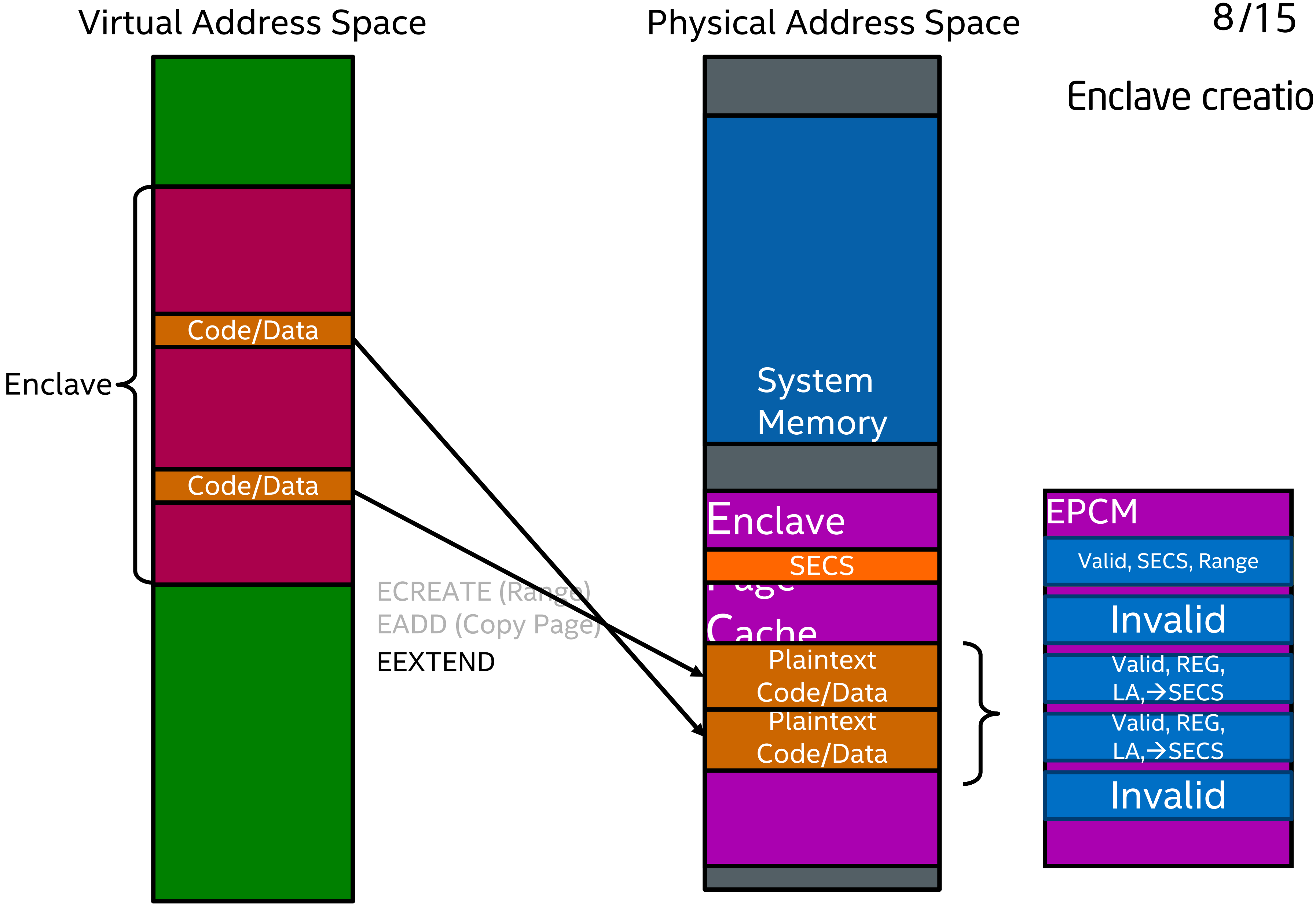
ECREATE (Range)
EADD (Copy Page)

Build

Life Cycle of An Enclave

8/15

Enclave creation



ECREATE (Range)
EADD (Copy Page)
EEXTEND

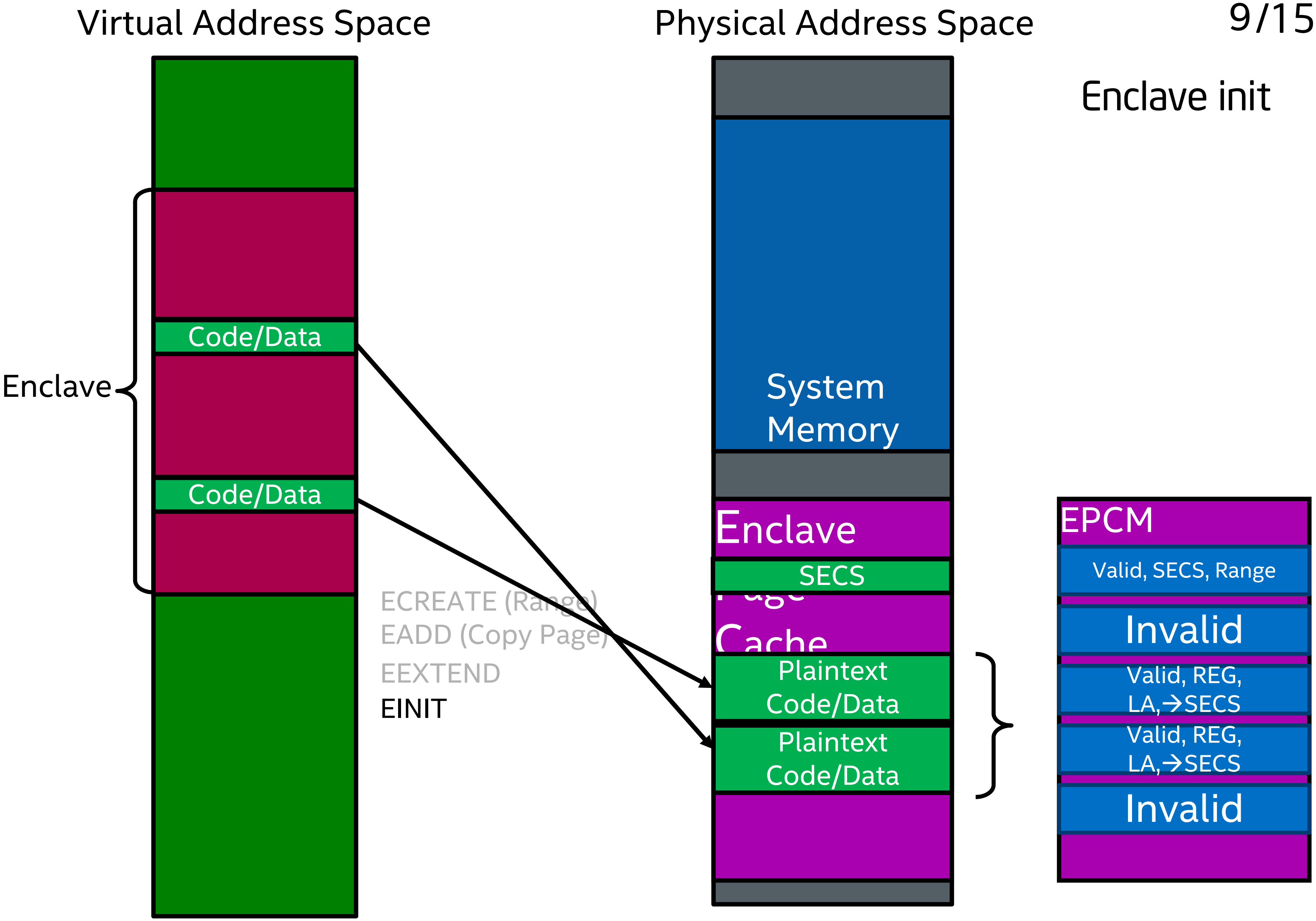
Intel and the Intel logo are trademarks of Intel Corporation in the U. S. and/or other countries. *Other names and brands may be claimed as the property of others. Copyright © 2015, Intel Corporation.

Build

Life Cycle of An Enclave

9/15

Enclave init



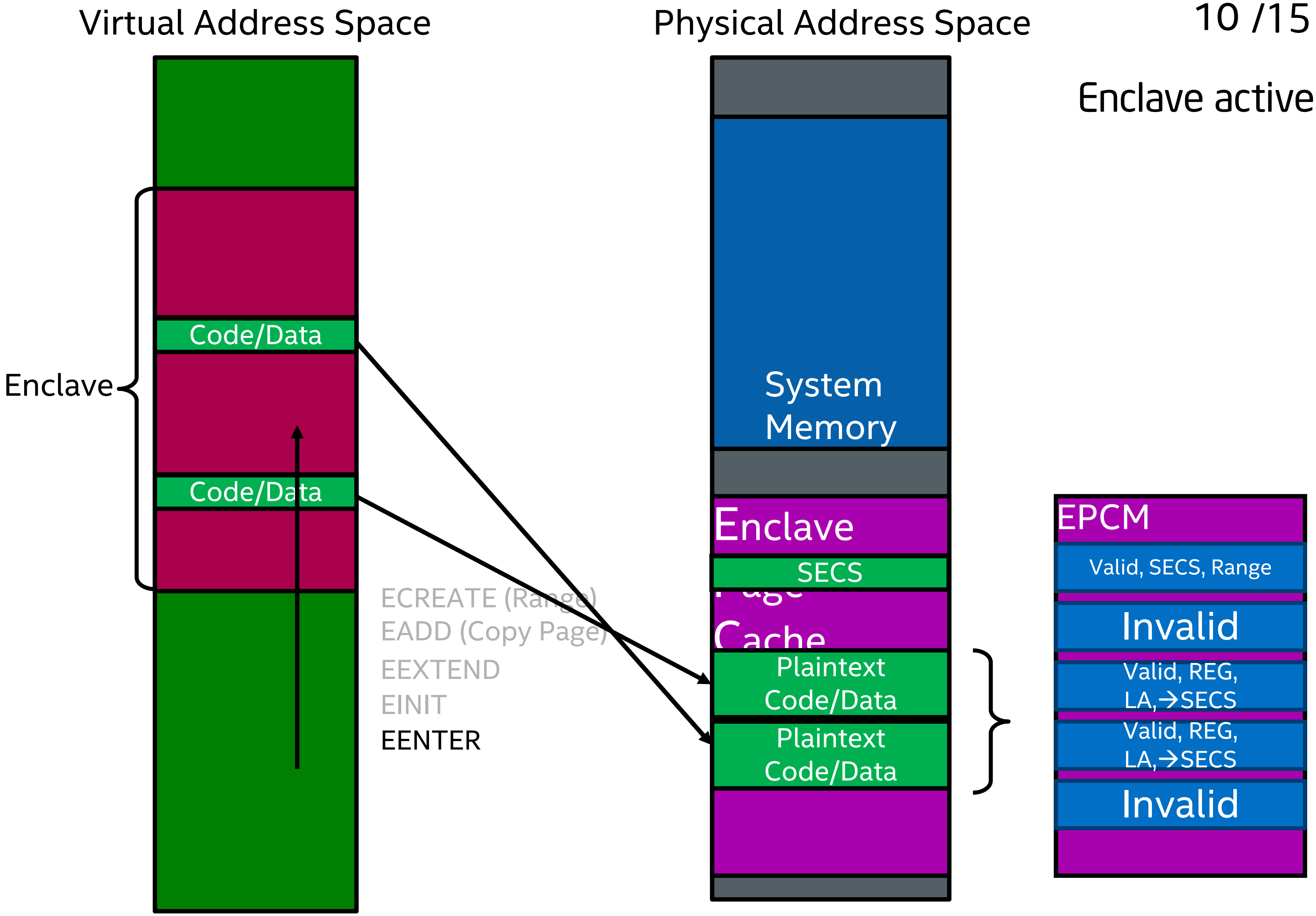
ECREATE (Range)
EADD (Copy Page)
EEXTEND
EINIT

Build

Life Cycle of An Enclave

10 /15

Enclave active



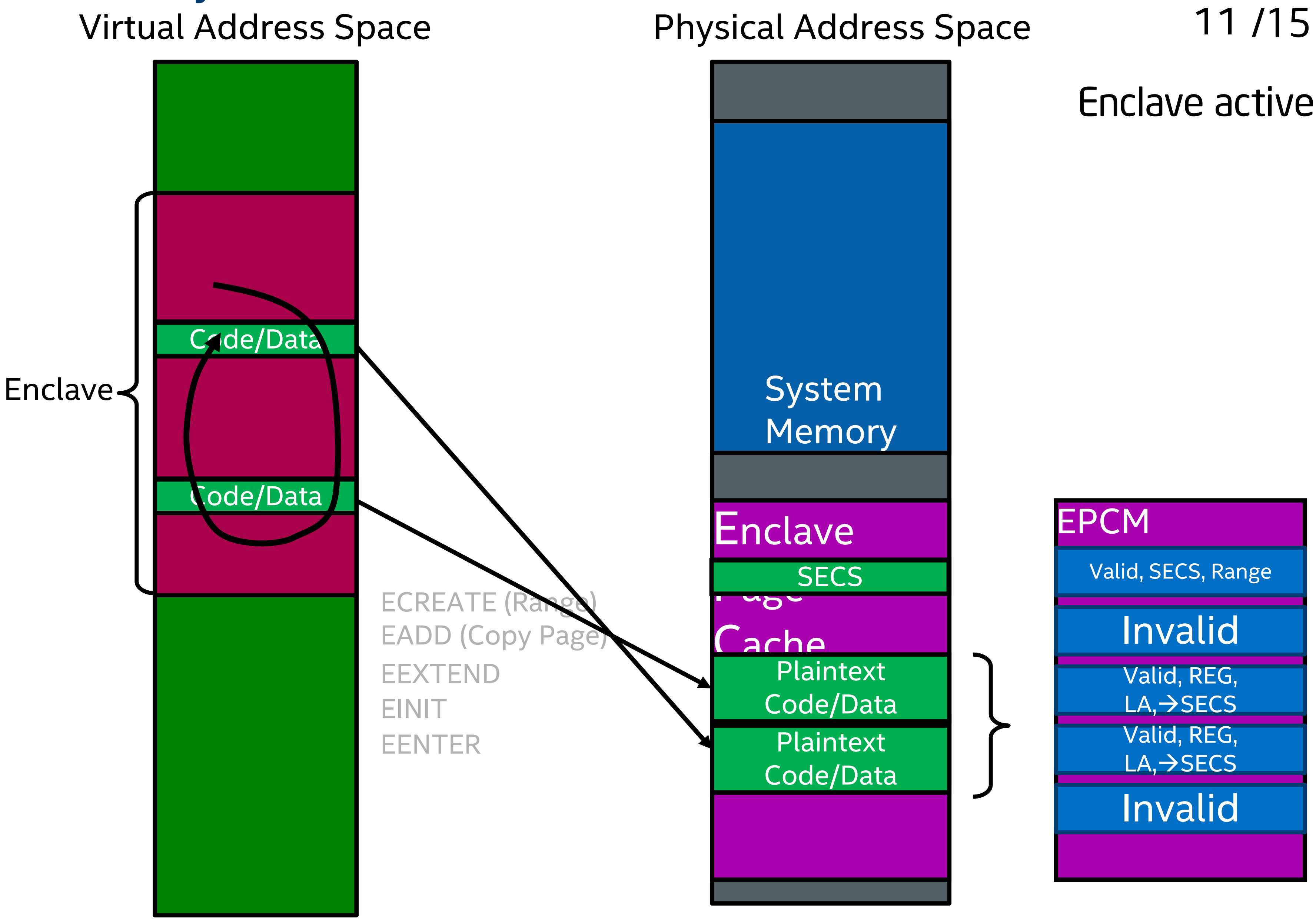
ECREATE (Range)
EADD (Copy Page)
EEXTEND
EINIT
EENTER

Build

Life Cycle of An Enclave

11 /15

Enclave active



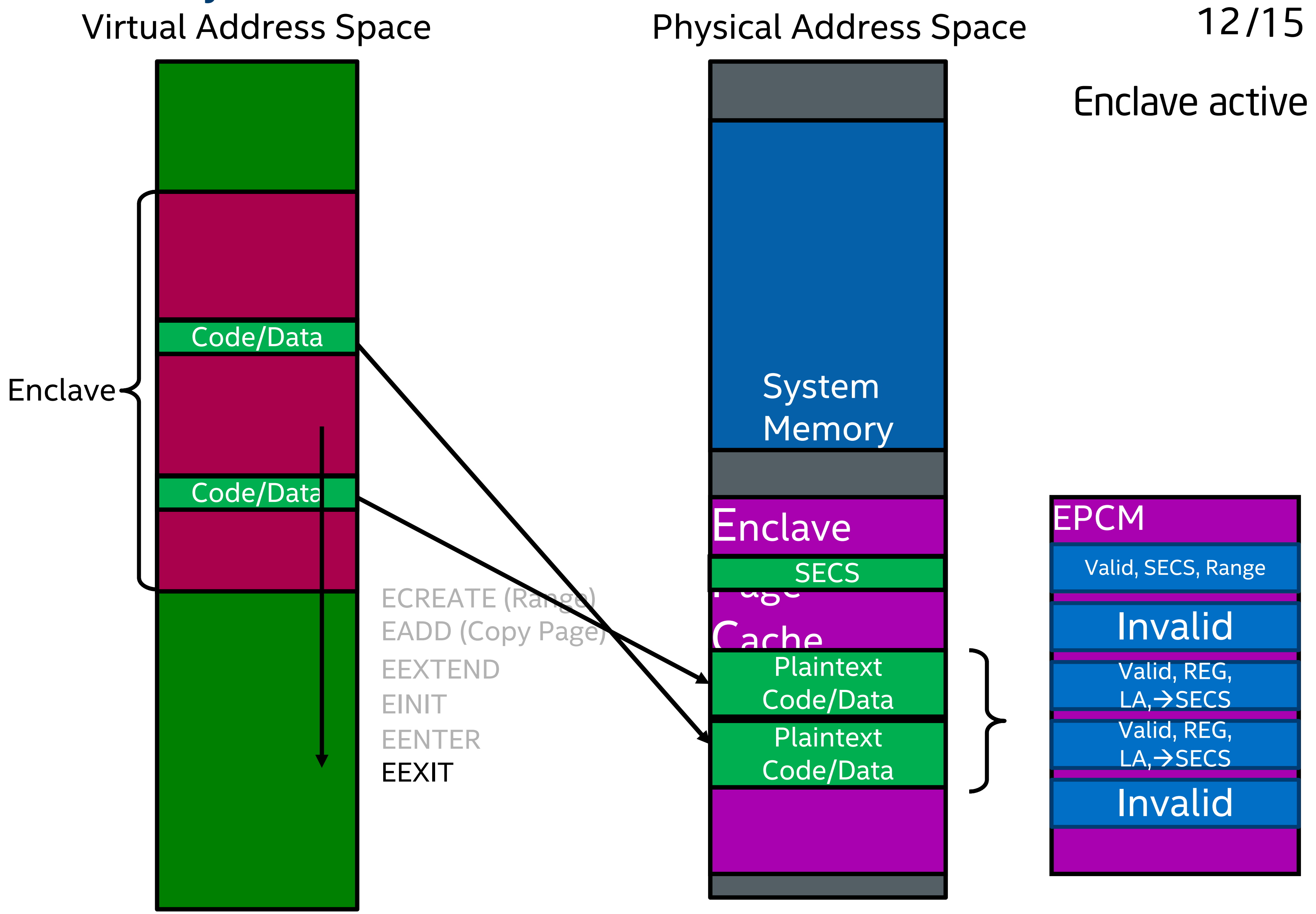
ECREATE (Range)
EADD (Copy Page)
EEXTEND
EINIT
EENTER

Build

Life Cycle of An Enclave

12/15

Enclave active



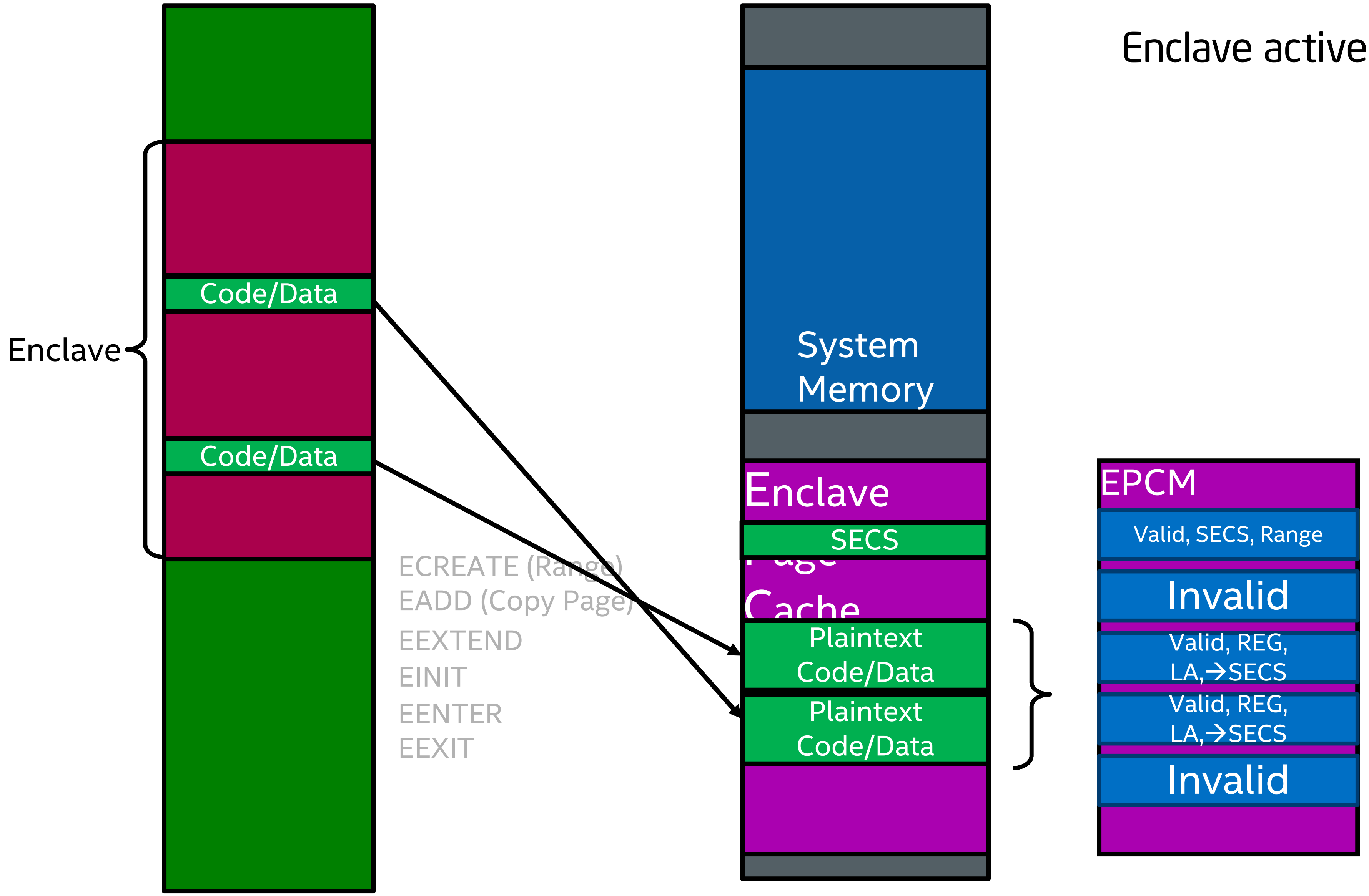
ECREATE (Range)
EADD (Copy Page)
EEXTEND
EINIT
EENTER
EEXIT

Build

Life Cycle of An Enclave

13 /15

Enclave active



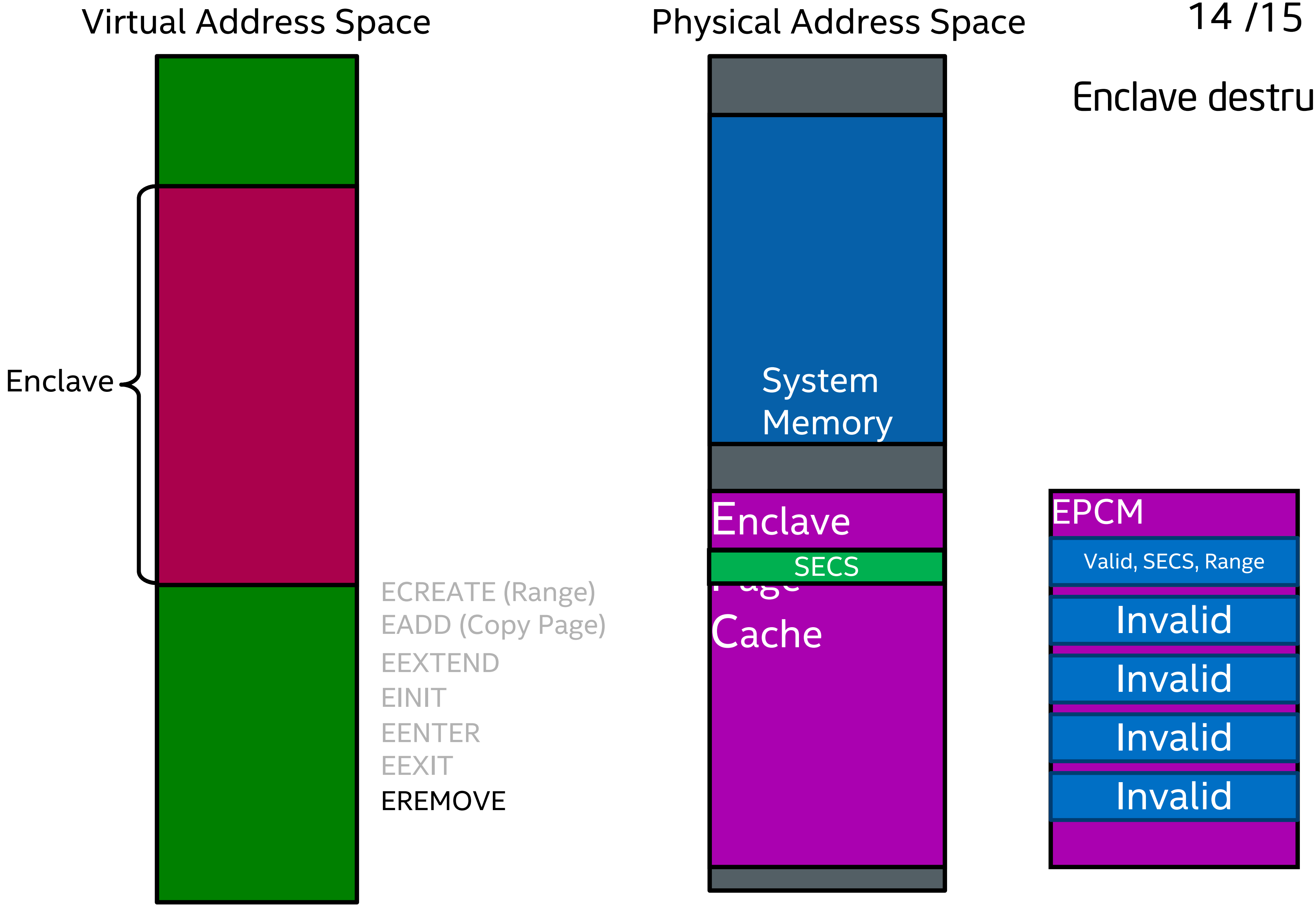
ECREATE (Range)
EADD (Copy Page)
EEXTEND
EINIT
EENTER
EEXIT

Build

Life Cycle of An Enclave

14 /15

Enclave destruction



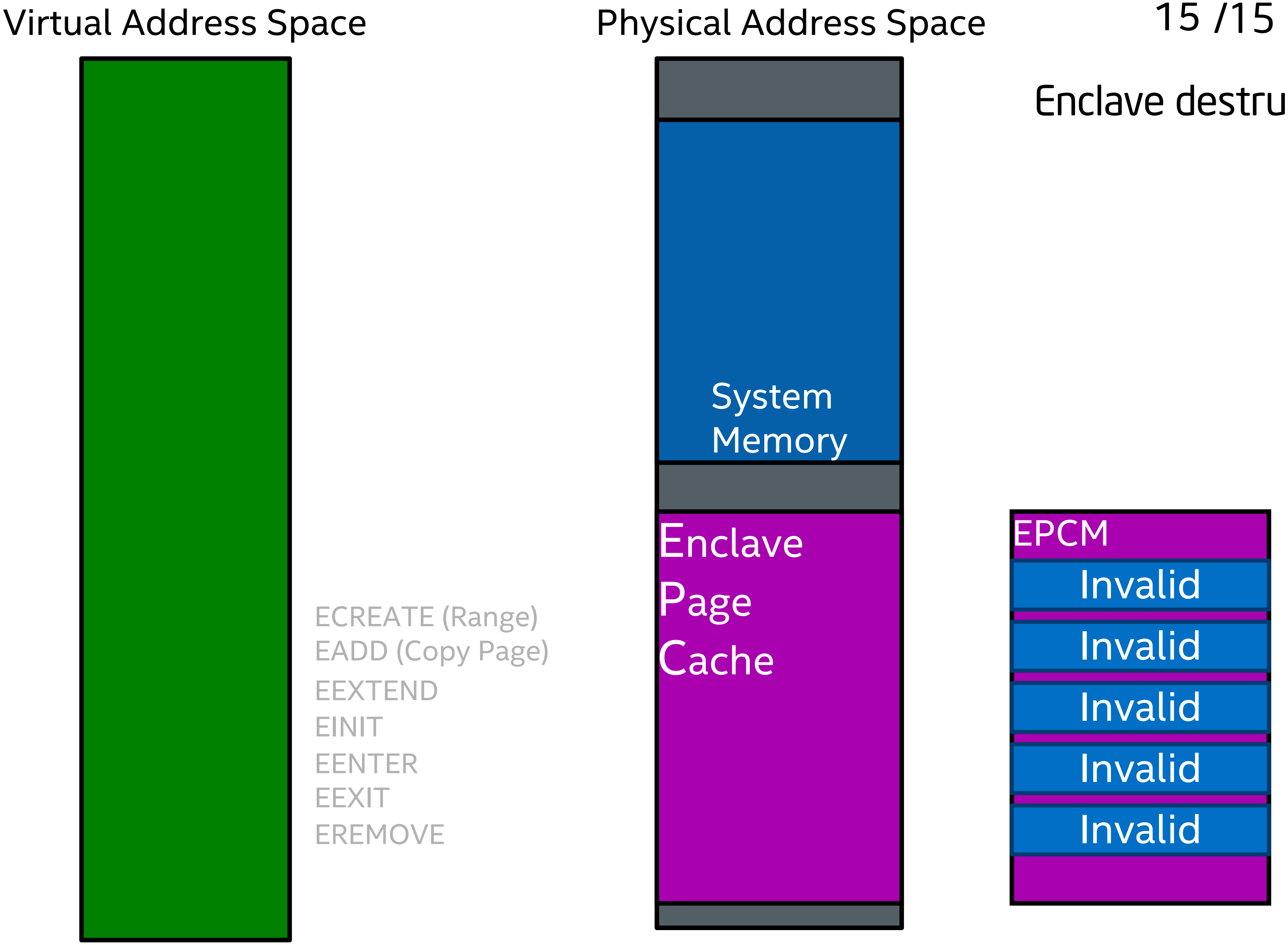
ECREATE (Range)
EADD (Copy Page)
EEXTEND
EINIT
EENTER
EEXIT
EREMOVE

Life Cycle of An Enclave

Build

15 /15

Enclave destruction



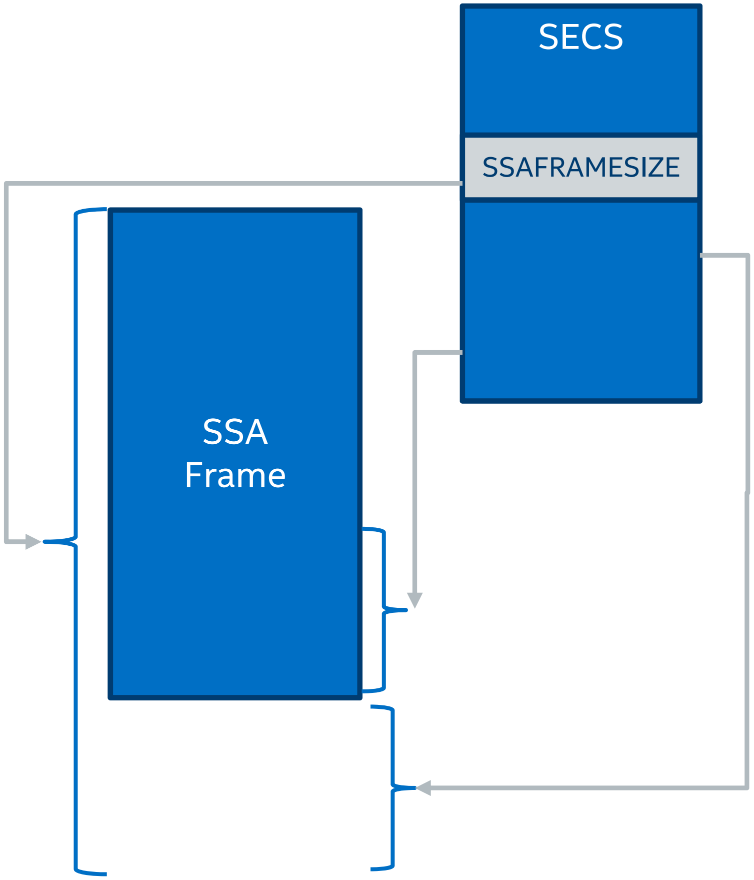
ECREATE (Range)
EADD (Copy Page)
EEXTEND
EINIT
EENTER
EEXIT
EREMOVE

Handling Exceptions

- Asynchronous Exit (AEX)
  - Faults, exceptions and interrupts initiate the Asynchronous Exit flow.
  - During AEX, enclave register state is stored in the enclave's active SSA frame and initialized to a known value prior to leaving the enclave
    - The RIP is initialized to an area referred to as the trampoline code
- SSA
  - Each enclave thread has a dedicated State Save Area frame entry that is pre-defined by the ISV for that thread

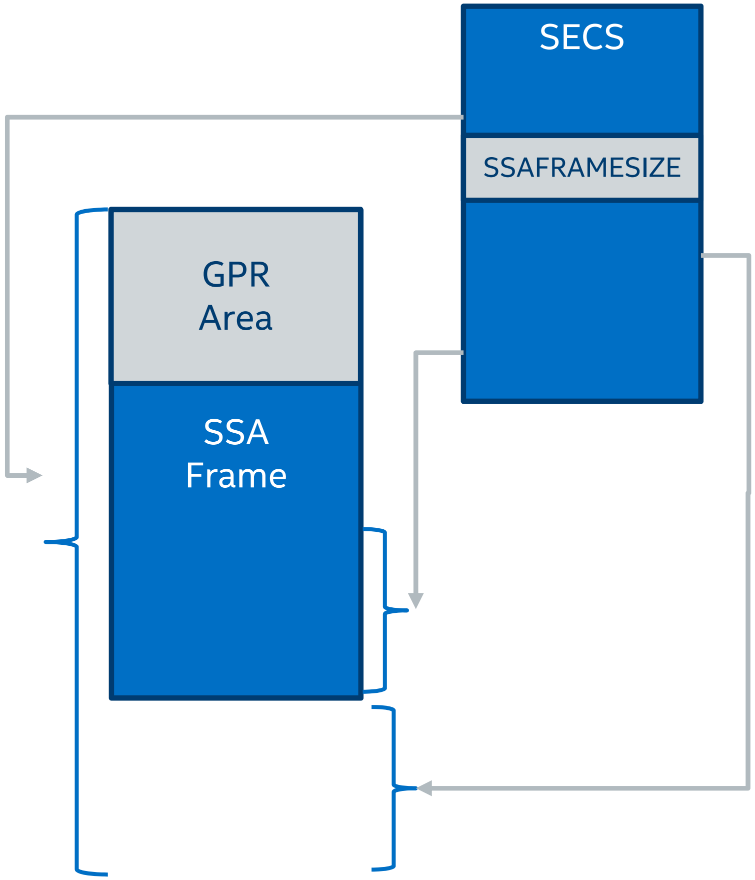
STATE SAVE Area (SSA) Frame

- When an asynchronous exit occurs while running in an enclave, the architectural state is saved in the SSA
    - SSAFRAMESIZE field in SECS defines size of the SSA frame (in pages)



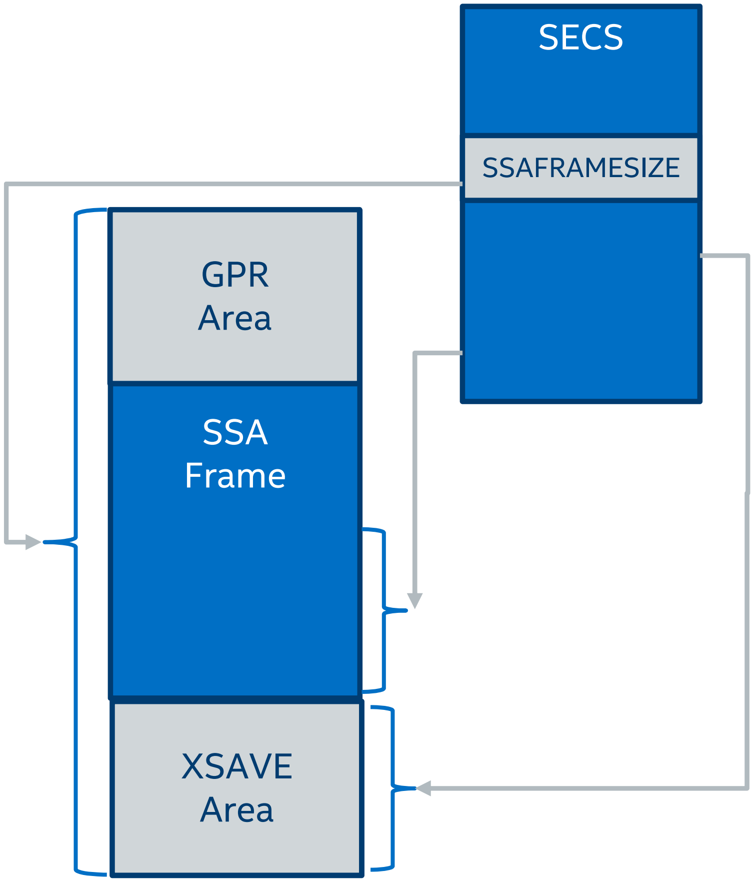
STATE SAVE Area (SSA) Frame

- When an asynchronous exit occurs while running in an enclave, the architectural state is saved in the SSA
  - SSAFRAMESIZE field in SECS defines size of the SSA frame (in pages)
  - GPRs and exception info is saved to fixed size GPR area of SSA frame on top of SSA frame (in VMCS format)



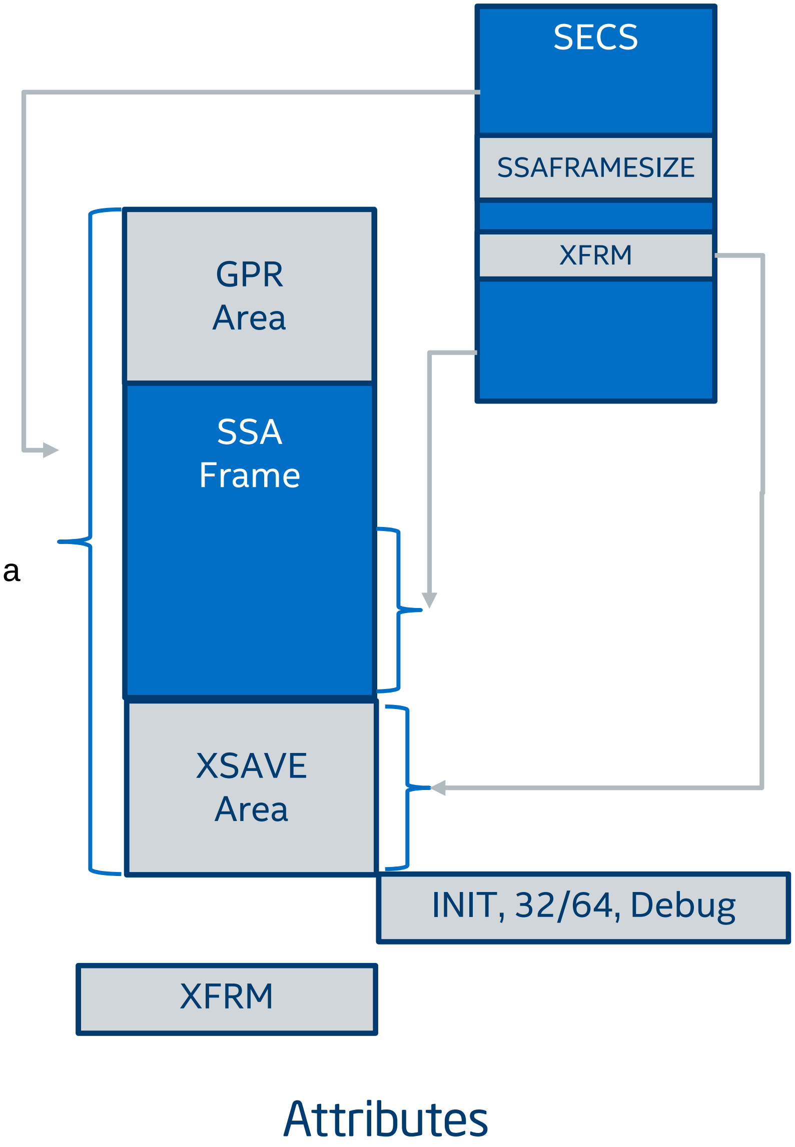
STATE SAVE Area (SSA) Frame

- When an asynchronous exit occurs while running in an enclave, the architectural state is saved in the SSA
  - SSAFRAMESIZE field in SECS defines size of the SSA frame (in pages)
  - GPRs and exception info is saved to fixed size GPR area of SSA frame on top of SSA frame (in VMCS format)
  - X87, SSE, AVX, etc. state is saved into variable size XSAVE area at bottom of SSA frame (in XSAVE format)



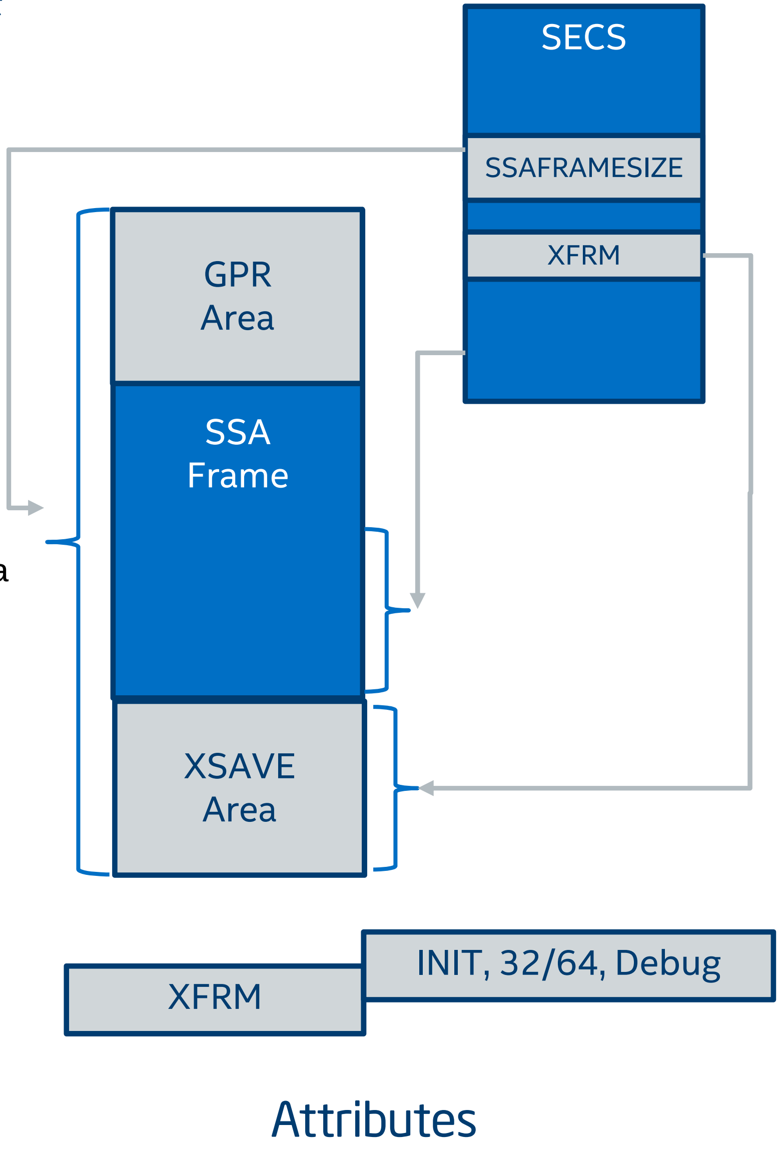
STATE SAVE Area (SSA) Frame

- When an asynchronous exit occurs while running in an enclave, the architectural state is saved in the SSA
  - SSAFRAMESIZE field in SECS defines size of the SSA frame (in pages)
  - GPRs and exception info is saved to fixed size GPR area of SSA frame on top of SSA frame (in VMCS format)
  - X87, SSE, AVX, etc. state is saved into variable size XSAVE area at bottom of SSA frame (in XSAVE format)
  - XFRM (XCR0 format) field in SECS controls size of XSAVE area and is contained in the upper bits of the SECS.ATTRIBUTES field defined by ECREATE instruction
  - MISC Area fields supported by CPU are enumerated by CPUID.SE_LEAF.0.EBX



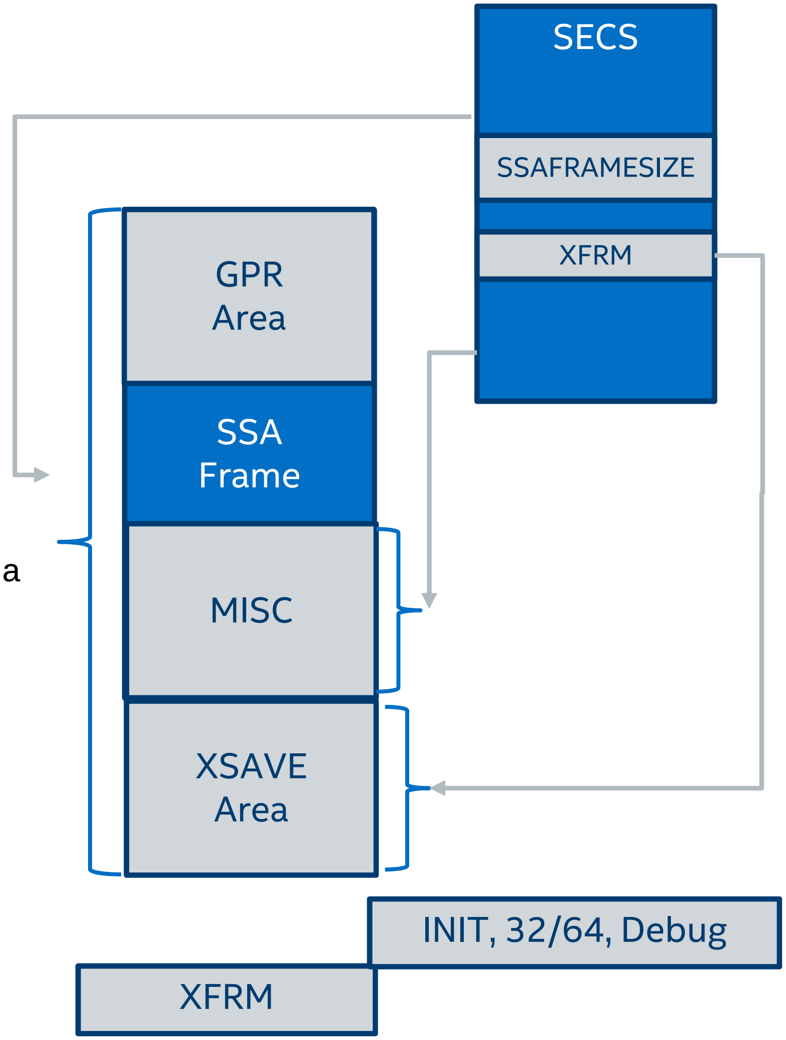
STATE SAVE Area (SSA) Frame

- When an asynchronous exit occurs while running in an enclave, the architectural state is saved in the SSA
  - SSAFRAMESIZE field in SECS defines size of the SSA frame (in pages)
  - GPRs and exception info is saved to fixed size GPR area of SSA frame on top of SSA frame (in VMCS format)
  - X87, SSE, AVX, etc. state is saved into variable size XSAVE area at bottom of SSA frame (in XSAVE format)
  - XFRM (XCR0 format) field in SECS controls size of XSAVE area and is contained in the upper bits of the SECS.ATTRIBUTES field defined by ECREATE instruction
  - MISC Area fields supported by CPU are enumerated by CPUID.SE_LEAF.0.EBX
- ECREATE checks that all areas fit within an SSA frame



STATE SAVE Area (SSA) Frame

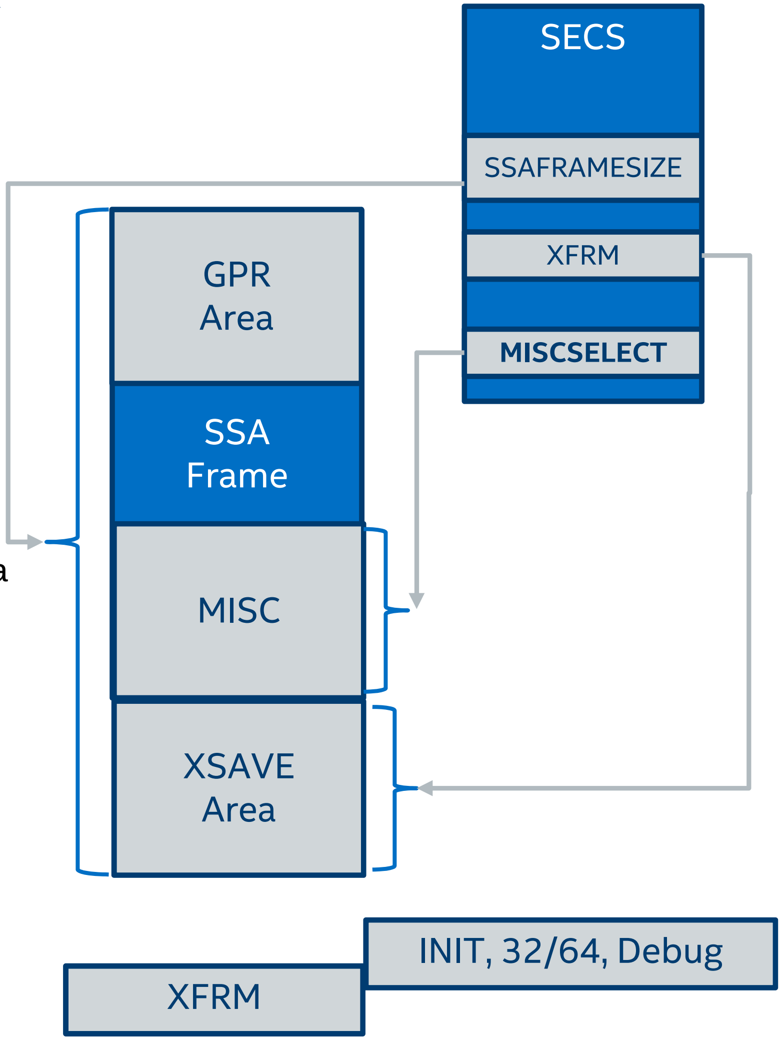
- When an asynchronous exit occurs while running in an enclave, the architectural state is saved in the SSA
  - SSAFRAMESIZE field in SECS defines size of the SSA frame (in pages)
  - GPRs and exception info is saved to fixed size GPR area of SSA frame on top of SSA frame (in VMCS format)
  - X87, SSE, AVX, etc. state is saved into variable size XSAVE area at bottom of SSA frame (in XSAVE format)
  - XFRM (XCR0 format) field in SECS controls size of XSAVE area and is contained in the upper bits of the SECS.ATTRIBUTES field defined by ECREATE instruction
  - MISC Area fields supported by CPU are enumerated by CPUID.SE_LEAF.0.EBX
- ECREATE checks that all areas fit within an SSA frame
- Each TCS maintains a stack of SSA frames
  - EENTER checks that an SSA frame is available, AEX increments current SSA frame pointer, ERESUME decrements current SSA frame pointer



Attributes

STATE SAVE Area (SSA) Frame

- When an asynchronous exit occurs while running in an enclave, the architectural state is saved in the SSA
  - SSAFRAMESIZE field in SECS defines size of the SSA frame (in pages)
  - GPRs and exception info is saved to fixed size GPR area of SSA frame on top of SSA frame (in VMCS format)
  - X87, SSE, AVX, etc. state is saved into variable size XSAVE area at bottom of SSA frame (in XSAVE format)
  - XFRM (XCR0 format) field in SECS controls size of XSAVE area and is contained in the upper bits of the SECS.ATTRIBUTES field defined by ECREATE instruction
  - MISC Area fields supported by CPU are enumerated by CPUID.SE_LEAF.0.EBX
- ECREATE checks that all areas fit within an SSA frame
- Each TCS maintains a stack of SSA frames
  - EENTER checks that an SSA frame is available, AEX increments current SSA frame pointer, ERESUME decrements current SSA frame pointer



Attributes

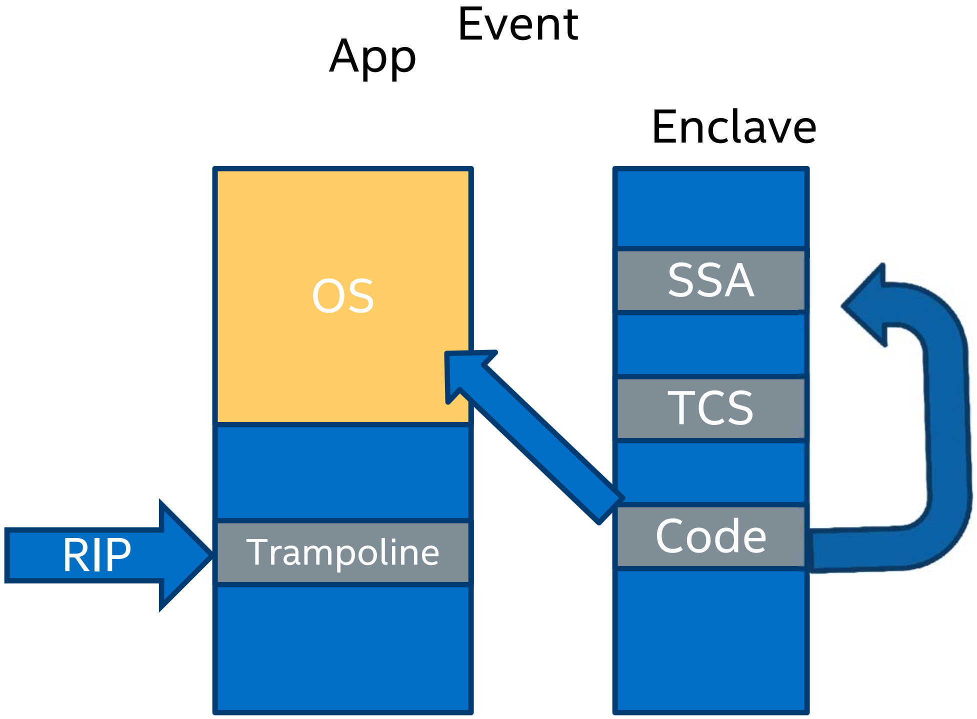
Resuming from exceptions

- Trampoline area
  - On an AEX the RIP is modified to point to the Trampoline area in the untrusted section of the app.
  - This RIP is pushed onto the stack when jumping to the OS handler
  - IRET will return the flow control to the app at the Trampoline Area
- ERESUME
  - The Trampoline will execute the ERESUME instruction
    - Register state will be restored from the SSA
    - Execution will resume from the interrupted location

Trampoline: Interrupts or Exceptions

Interrupts and exceptions inside enclave cause exit to OS

- Processor state saved inside enclave's save state area (SSA).
- Synthetic state and trampoline's return address loaded into regs
- OS sees interrupts and exceptions just like today



Save Registers and RIP

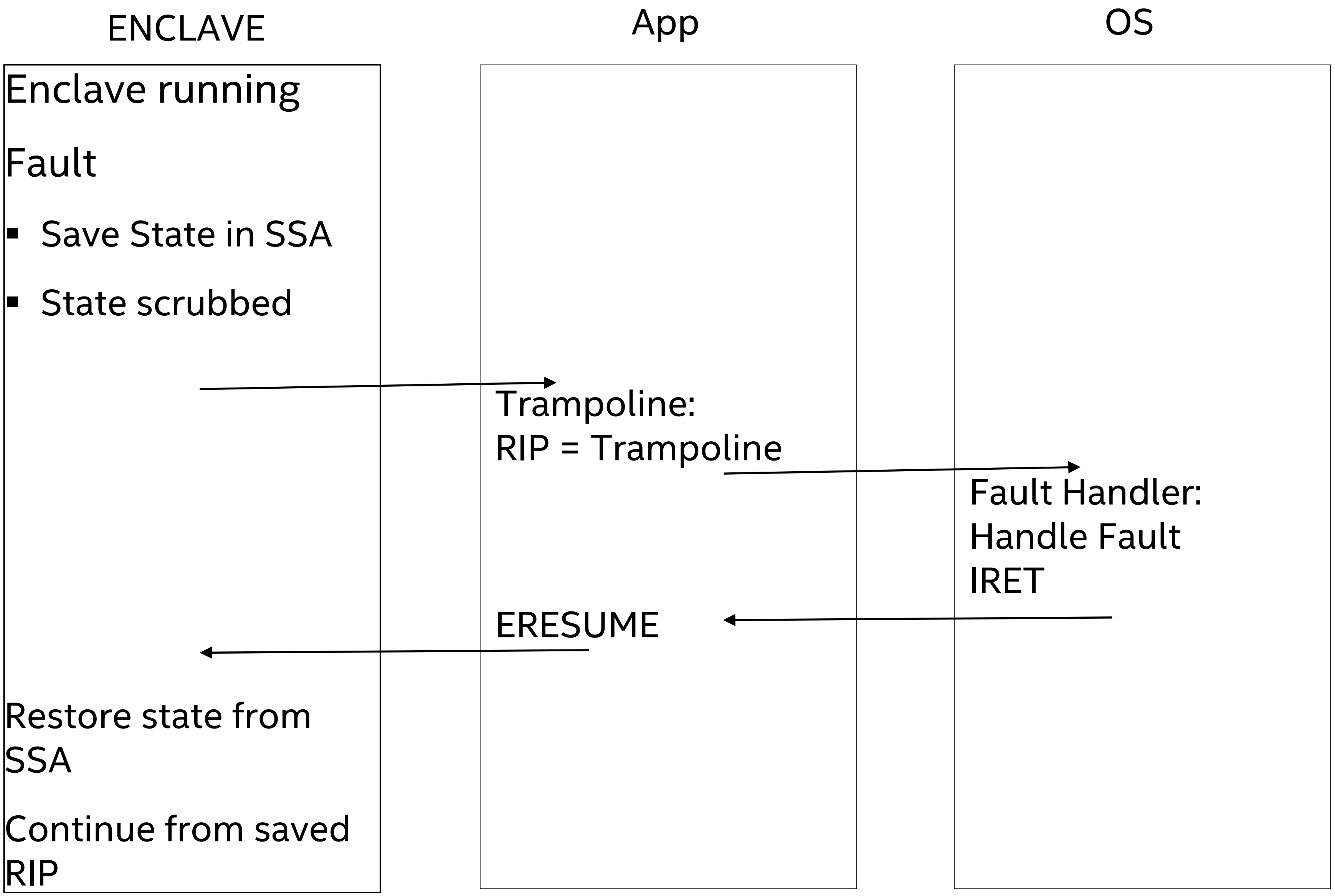
Return address is trampoline

Push exception info onto stack

Nesting of exceptions

- SGX supports nesting of exceptions
  - TCS can define multiple SSA frames on the SSA stack
- SSA frame pointer management
  - AEX pushes content onto the SSA frame and increments the SSA frame pointer
  - ERESUME decrements the SSA frame pointer and pops the content from the SSA frame into register state
  - EENTER leaves the pointer intact
    - be used to enter a trusted exception handler

Handling Exceptions



ENCLAVE

App

OS

Enclave running

Fault

- Save State in SSA
- State scrubbed

Trampoline:
RIP = Trampoline

Fault Handler:
Handle Fault
IRET

ERESUME

Restore state from SSA

Continue from saved RIP

EPC Page Swapping

- EPC memory is set by BIOS and limited from size perspective
- We need a way to remove an EPC page, place into unprotected memory, and restore it later.
- Page must maintain same security properties (confidentiality, anti-replay, and integrity) when restored
- EPC paging instructions provide ability to encrypt page and produce meta data needed to meet requirements
- An enclave page must be evicted only after all cached translations to that page have been evicted from all logical processors.
- Content is swapped on 4KByte page basis
  - Each 4KByte EPC page produces
    - 4KByte of encrypted content
    - 128Byte of meta-data (PCMD).

Paging Operations at a High Level

When a page is evicted from EPC

- It is assigned a unique version number which is recorded in a new type of EPC page called version array (VA)
- Encrypted page, metadata, and EPCM information are written out to system memory

When page is reloaded

- The processor decrypts, and integrity checks the page, using crypto metadata
- The processor verifies that version is the same version that was last written out

EPC Paging instruction - I

- EPA
  - Allocates a 4KByte page in EPC for holding an array of page versions (VA) for anti-replay protection
    - VA contains versions of paged out enclave pages, size of each version slot is 64 bits.
- EBLOCK
  - Blocks a page from being accessed in preparation for swapping it out
    - Any future accesses by owner enclave to BLOCKED page result in #PF
    - Returns indication that page previously blocked
- ETRACK
  - Sets a tracking mechanism to verify that all TLB entries for the blocked page has been flushed

EPC Paging instruction - II

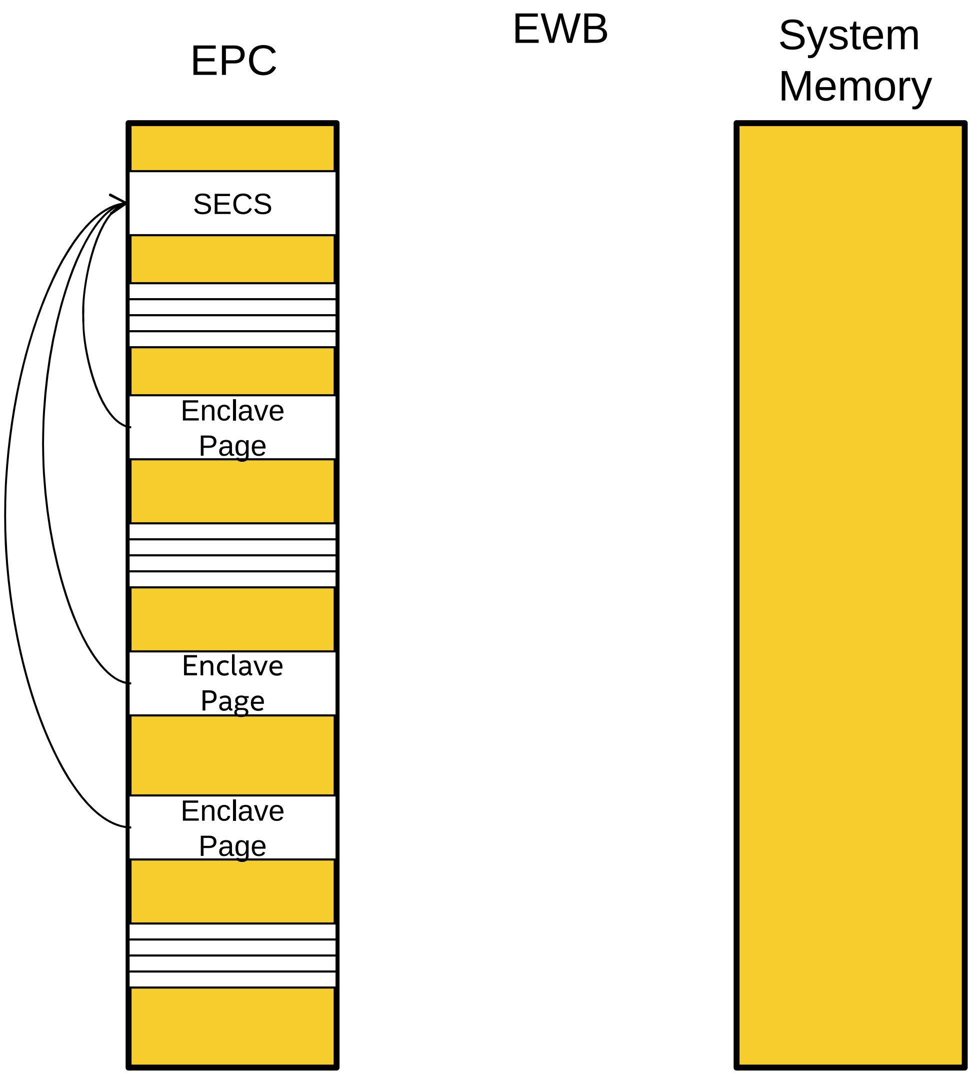
- EWB
  - Enclave page must be prepared for eviction: blocked and no TLB entry refer to that page.
  - Securely evicts a 4KByte page from the EPC along with it's page information
    - Assigning a unique version value for the page and storing it in the VA page.
    - Encrypt EPC page, create MAC over the encrypted page, version counter, and meta data. And write it out to external memory

EPC Paging instruction - III

- ELDU/B
  - Securely loads a page back from memory into the EPC into an unblocked or blocked state
    - Verify the MAC on the meta data, version counter from specific VA entry, and encrypted enclave page content
    - If verification succeed, decrypt the enclave page content into EPC page allocated by system memory and clear the VA entry.

BUILD

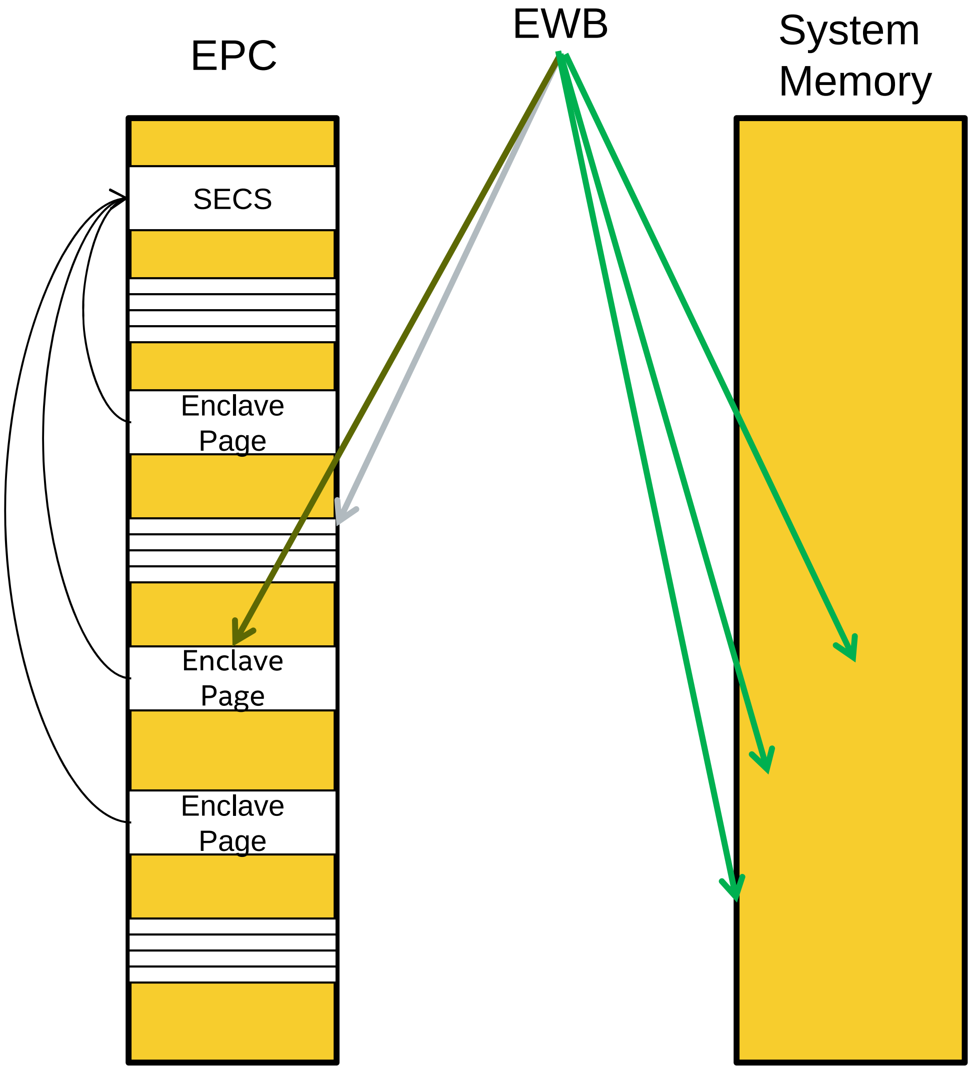
Page-out Example



Page-out Example

BUILD

1



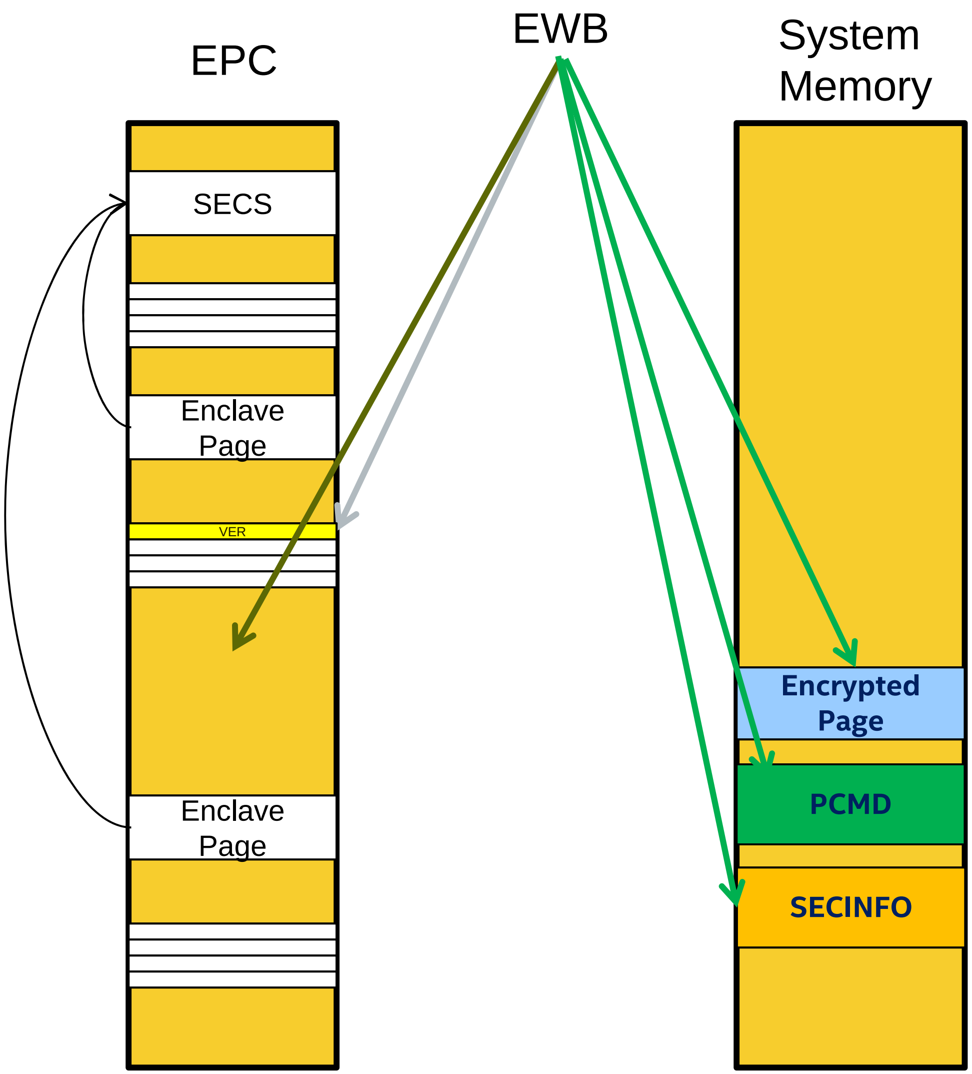
EWB Parameters:

- Pointer to EPC page that needs to be paged out
- Pointer to empty version slot
- Pointers outside EPC location

BUILD

2

Page-out Example



EWB Parameters:

- Pointer to EPC page that needs to be paged out
- Pointer to empty version slot
- Pointers outside EPC location

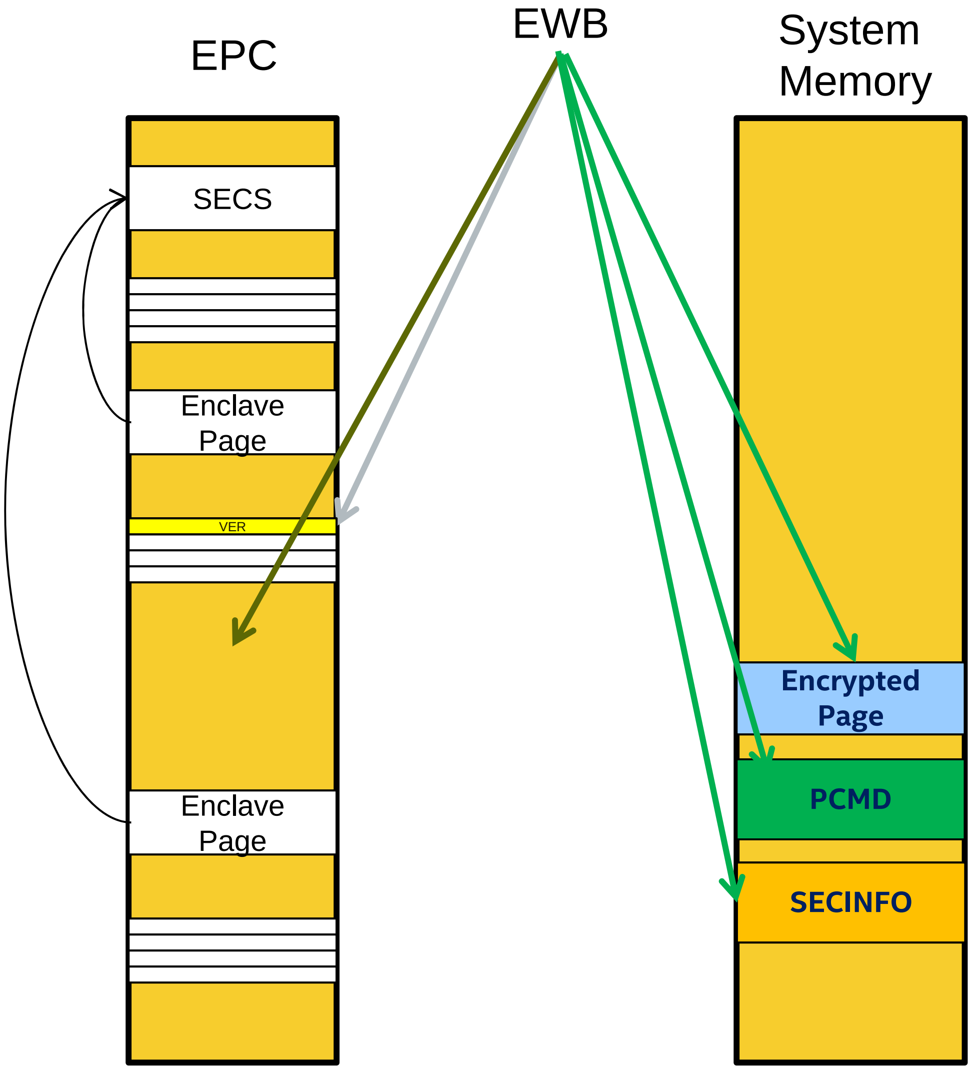
EWB Operation

- Remove page from the EPC
- Populate version slot
- Write encrypted version to outside

BUILD

3

Page-out Example



EWB Parameters:

- Pointer to EPC page that needs to be paged out
- Pointer to empty version slot
- Pointers outside EPC location

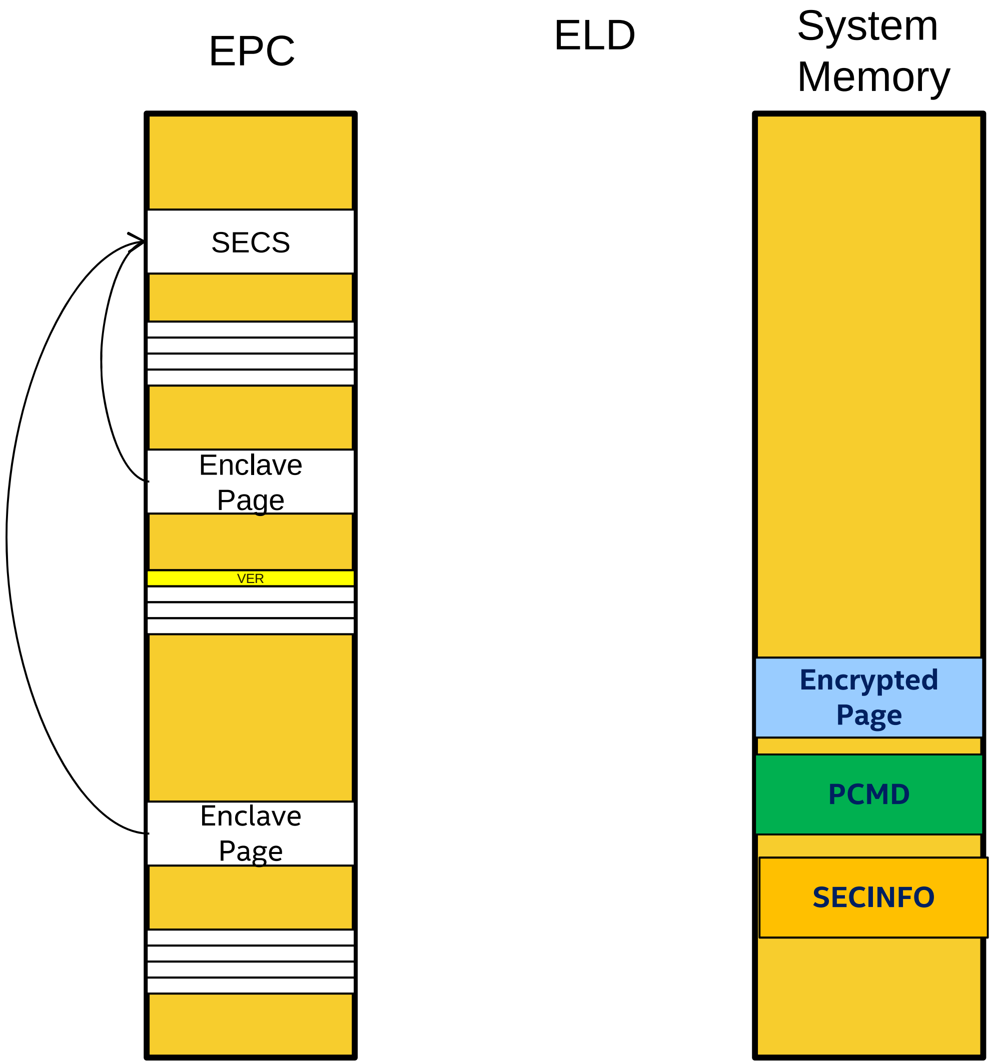
EWB Operation

- Remove page from the EPC
- Populate version slot
- Write encrypted version to outside

All pages, including SECS and Version Array can be paged out

BUILD

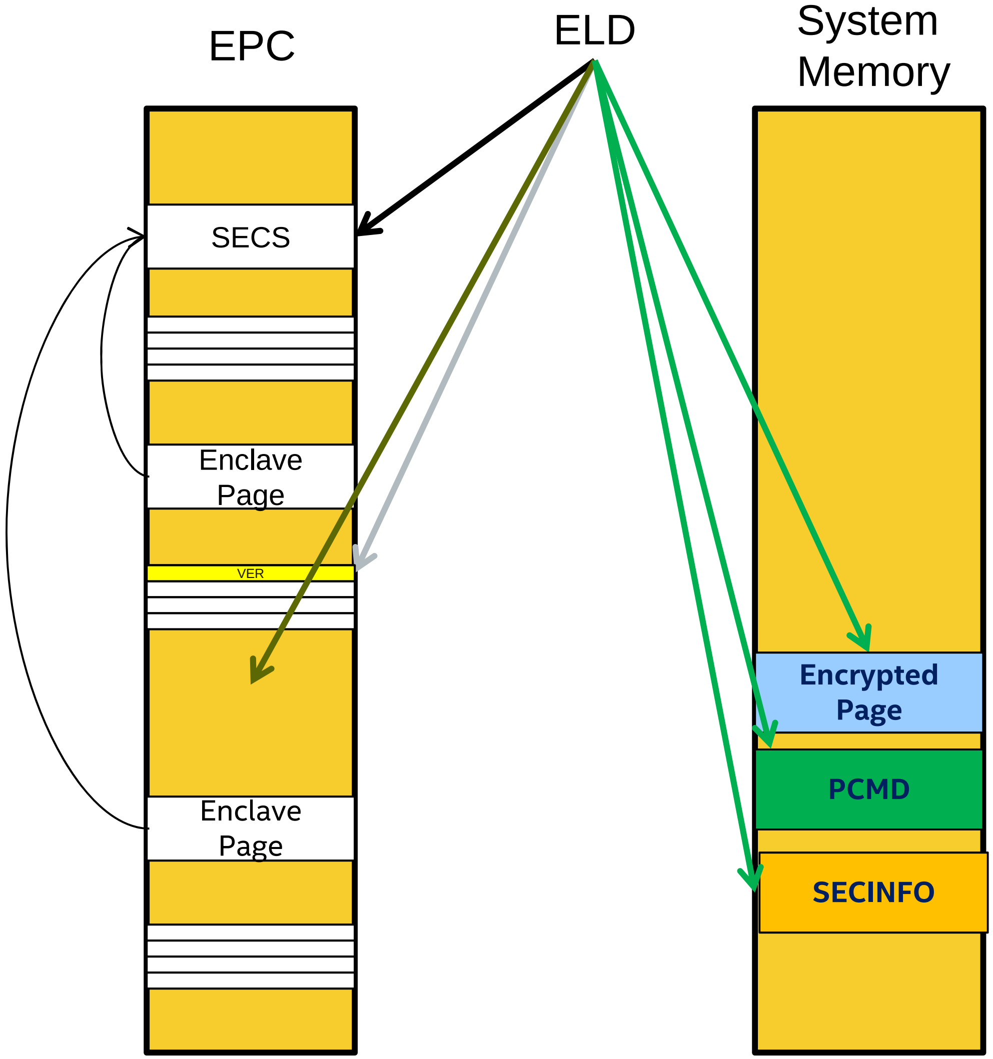
Page-in Example



Page-in Example

BUILD

1



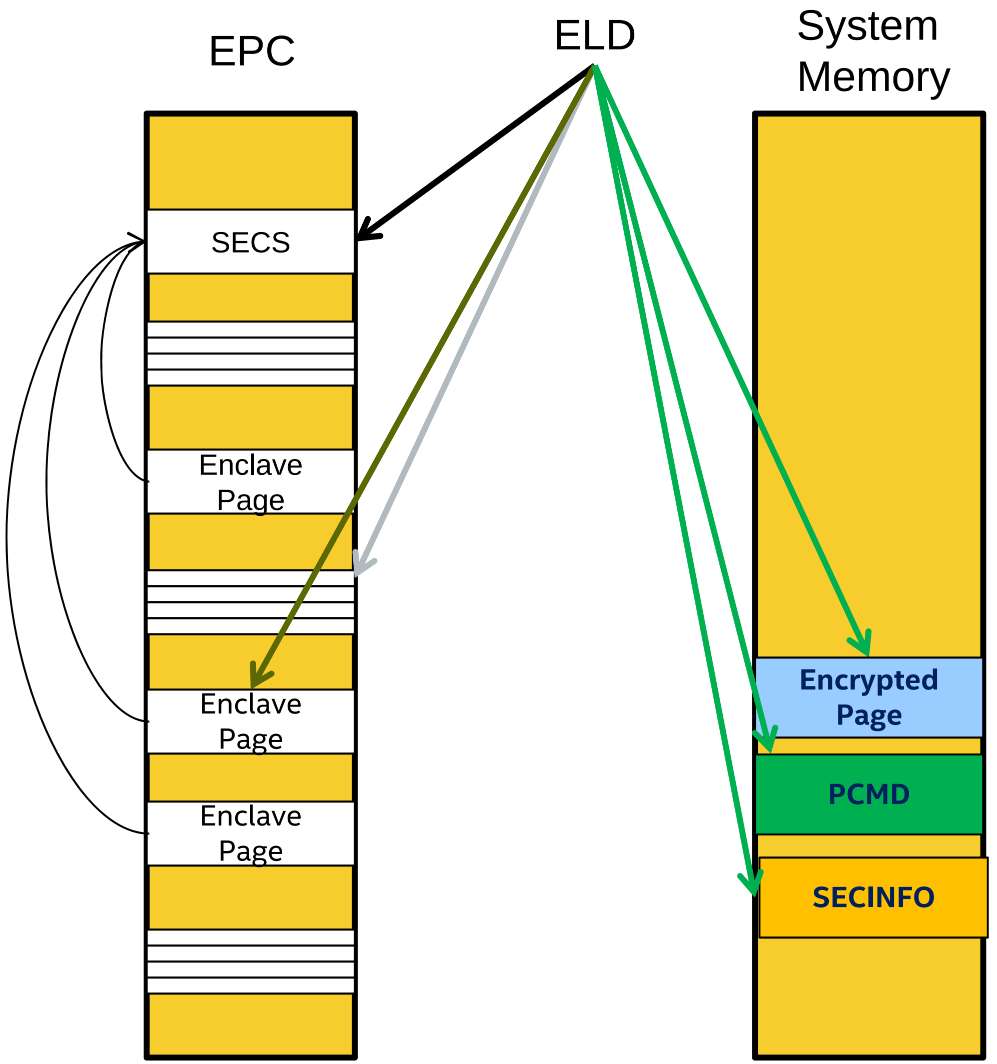
ELD Parameters:

- Encrypted page
- Free EPC page
- SECS (for an enclave page)
- Populated version slot

BUILD

2

Page-in Example



ELD Parameters:

- Encrypted page
- Free EPC page
- SECS (for an enclave page)
- Populated version slot

ELD Operation

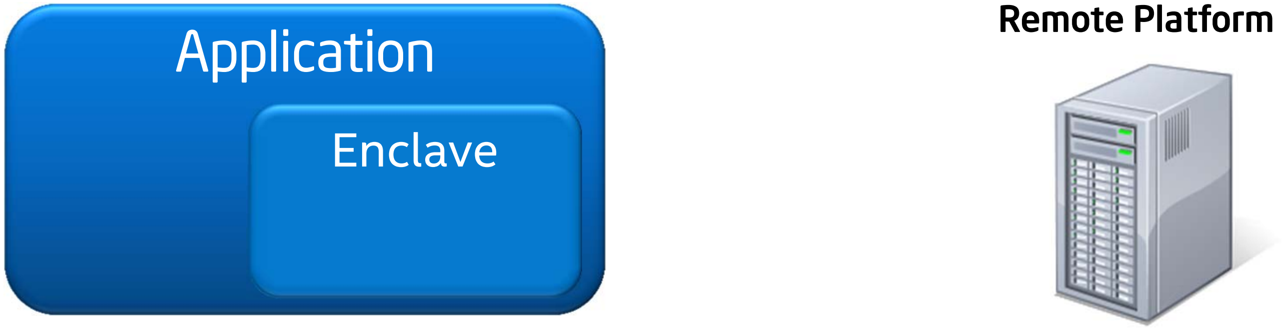
- Verify and decrypt the page using version
- Populate the EPC slot
- Free-up version slot

SGX Key Hierarchy, Attestation, Provisioning

The Challenge – Provisioning Secrets to the Enclave

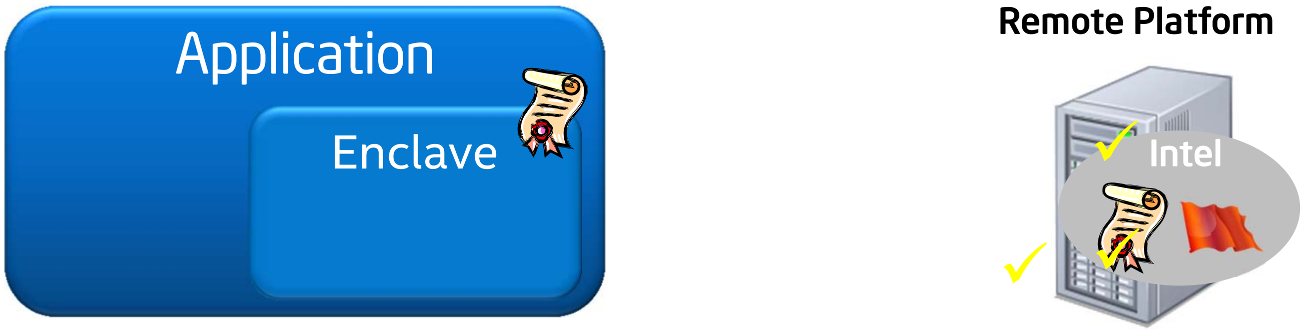
- An enclave is in the clear before instantiation
  - Sections of code and data could be encrypted, but their decryption key can't be pre-installed
- Secrets come from outside the enclave
  - Keys
  - Passwords
  - Sensitive data
- The enclave must be able to convince a 3rd party that it's trustworthy and can be provisioned with the secrets
- Subsequent runs should be able to use the secrets that have already been provisioned

Critical Features: Attestation and Sealing



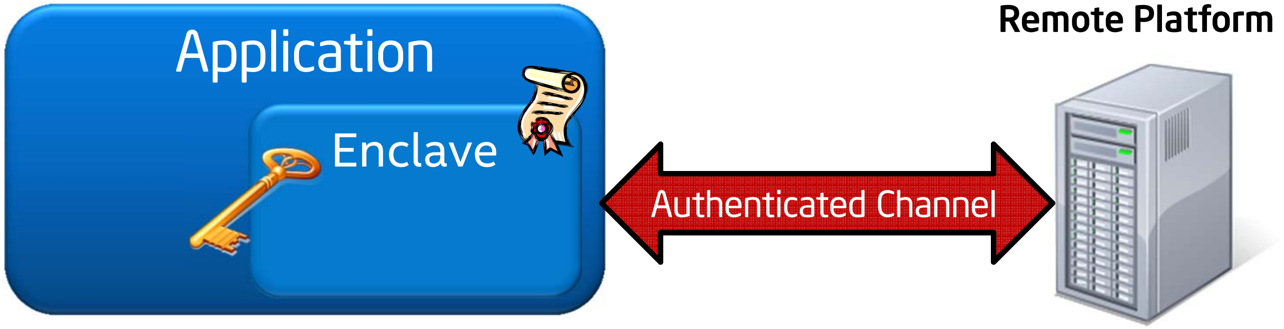
- App executes on local platform

Critical Features: Attestation and Sealing



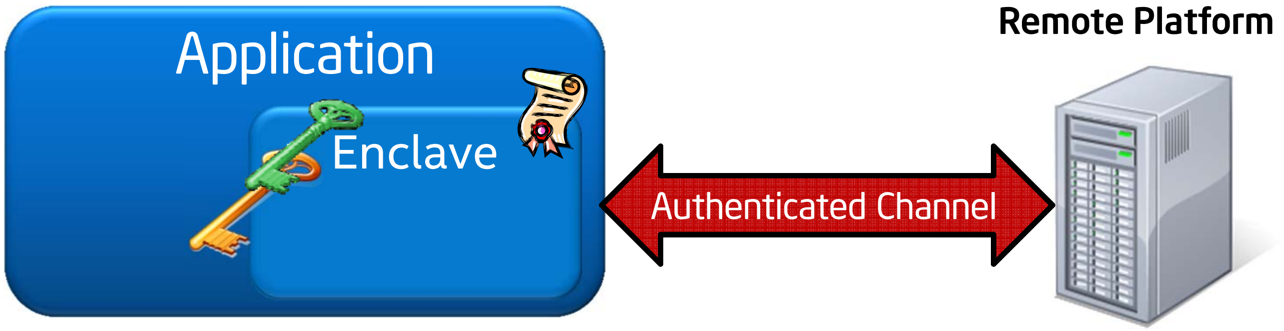
- App executes on local platform
- HW based **Attestation** provides remote platform assurance that “this is the right app executing in the right platform “

Critical Features: Attestation and Sealing



- App executes on local platform
- HW based **Attestation** provides remote platform assurance that “this is the right app executing in the right platform ”
  ⇨Remote platform can provision local platform with secrets

Critical Features: Attestation and Sealing



- App executes on local platform
- HW based **Attestation** provides remote platform assurance that “this is the right app executing in the right platform ”
  ⇨Remote platform can provision local platform with secrets
- App can seal secrets to platform for future use

Trustworthiness

- Intel® SGX supports enclave attestation to a 3rd party:
  - What software is running inside the enclave
  - Which execution environment the enclave is running at
  - Which Sealing Identity will be used by the enclave
  - What's the CPU's security level

- This accomplished via a combination of SGX instructions that supports local attestation and an Intel provided attestation enclave to support remote attestation.

- Once provisioned a secret, an enclave can seal its data using SGX instructions

SGX Key Hierarchy

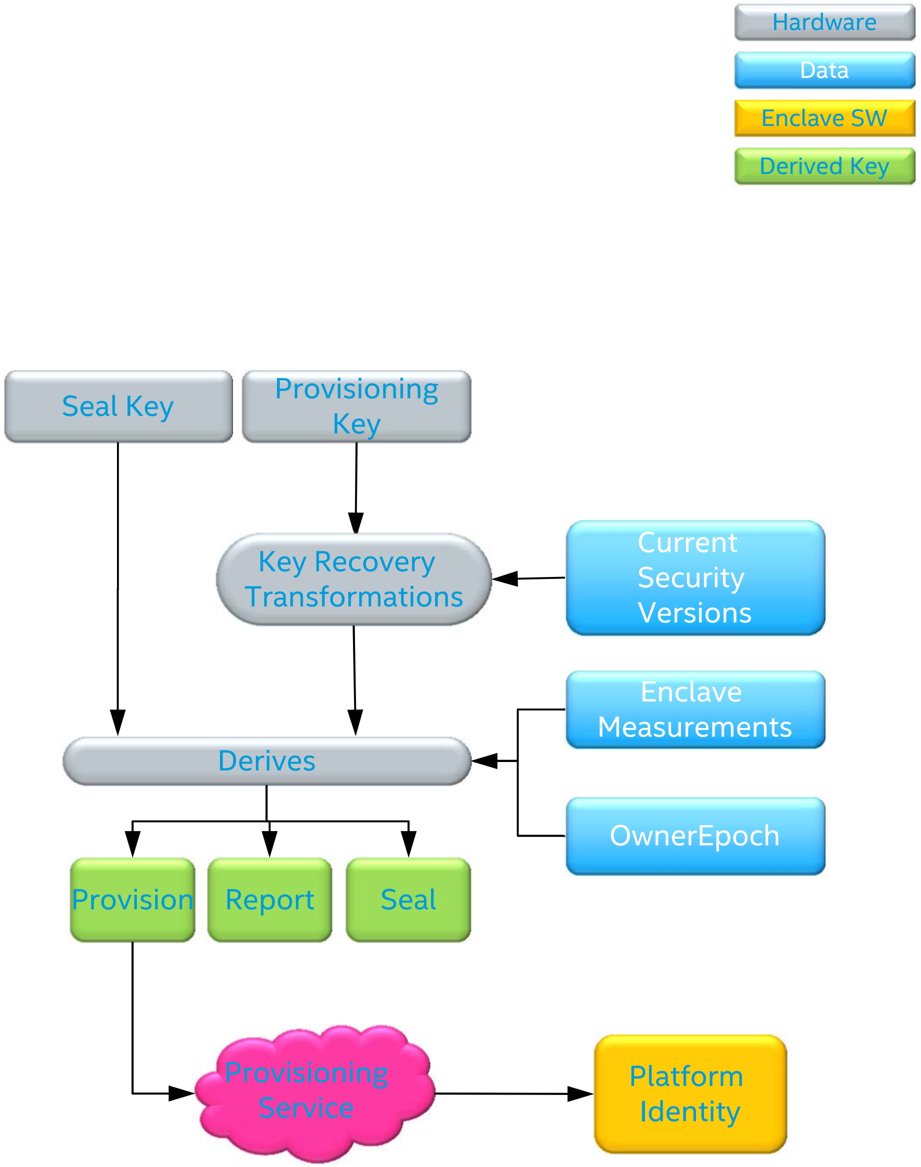
Platform unique keys are made available to enclave software via EGETKEY

SGX offers various keys for different types of operations (attestation vs data protection)

EGETKEY uses a key derivation algorithm to generate enclave specific keys

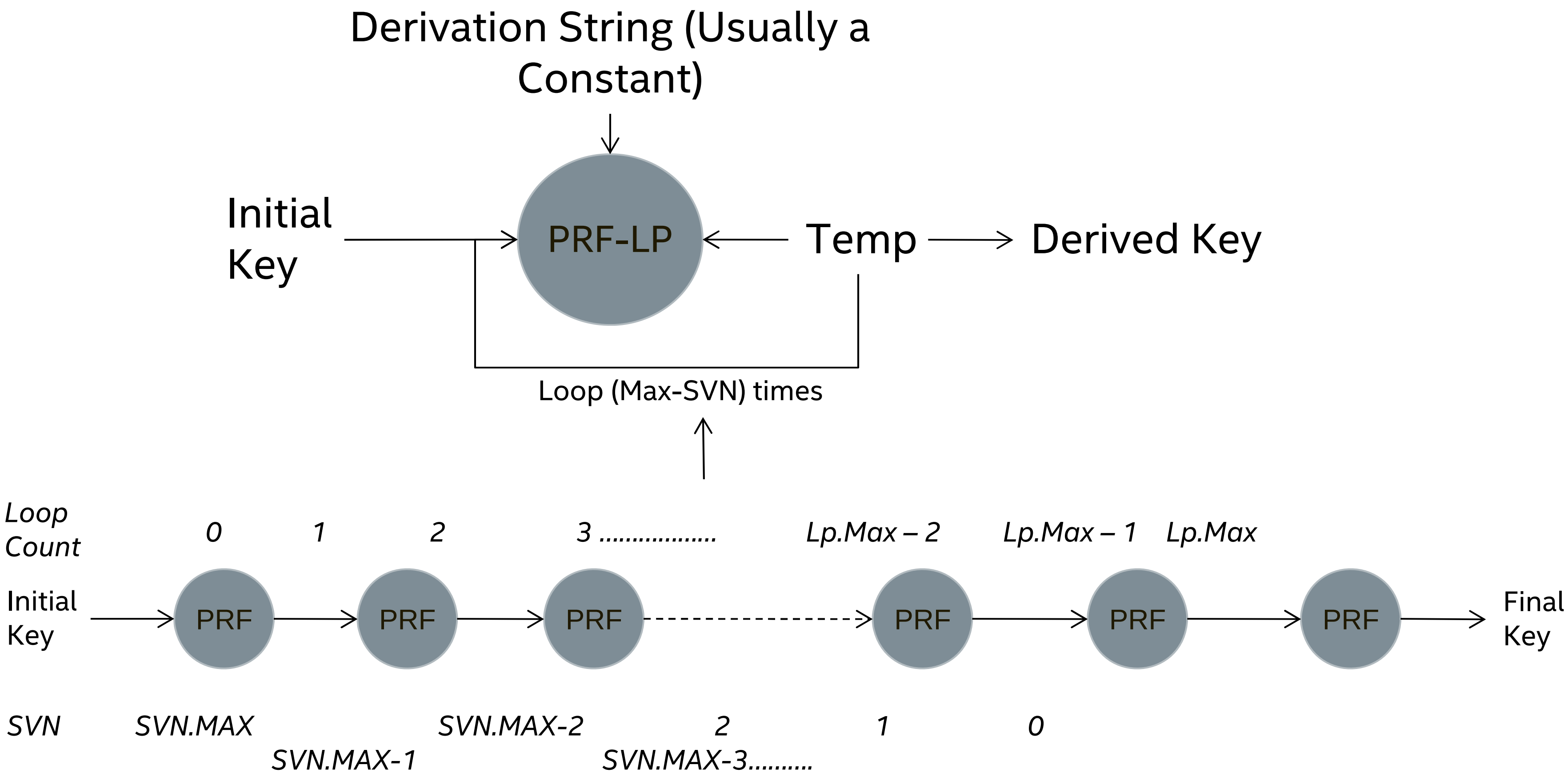
Inputs to this algorithm are:

- enclave’s identity,
- OwnerEpoch,
- Key type.



Key Recovery Transformation

Goal: Calculate older keys from newer keys, but not reverse.



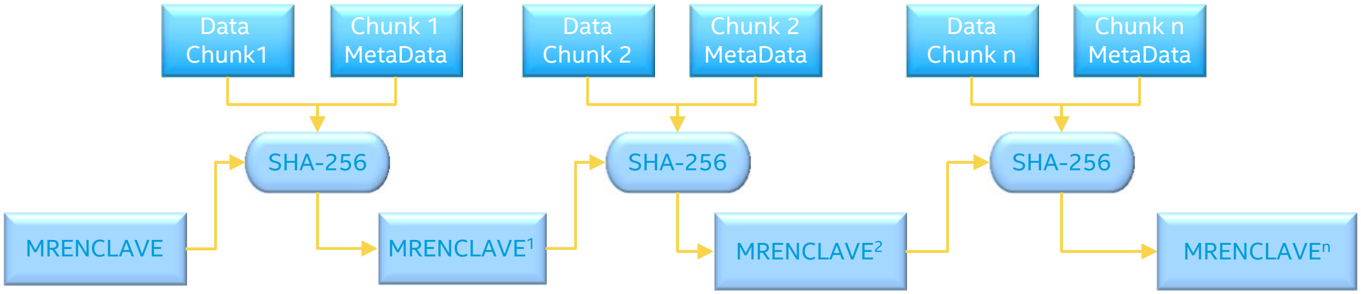
SGX Measurement

When building an enclave, Intel® SGX generates a cryptographic log of all the build activities

- Content: Code, Data, Stack, Heap
- Location of each page within the enclave
- Security flags being used

MRENCLAVE ("Enclave Identity") is a 256-bit digest of the log

- Represents the enclave's software TCB



Attestation

SGX provides LOCAL and REMOTE attestation capabilities

Local attestation allows one enclave to attest its TCB to another enclave on the same platform

Remote attestation allows which one enclave to attest its TCB to another entity outside of the platform

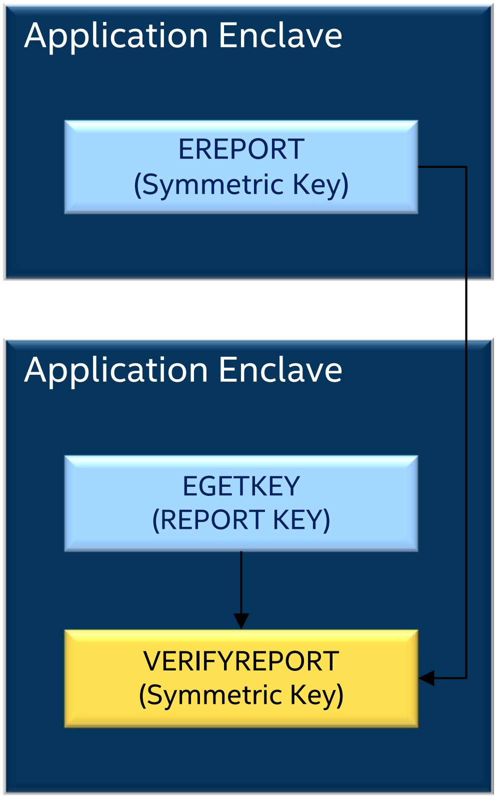
Hardware

Software

Local Attestation

SGX provides a hardware assertion, REPORT, that contains calling enclaves Attributes, Measurements and User supplied data.

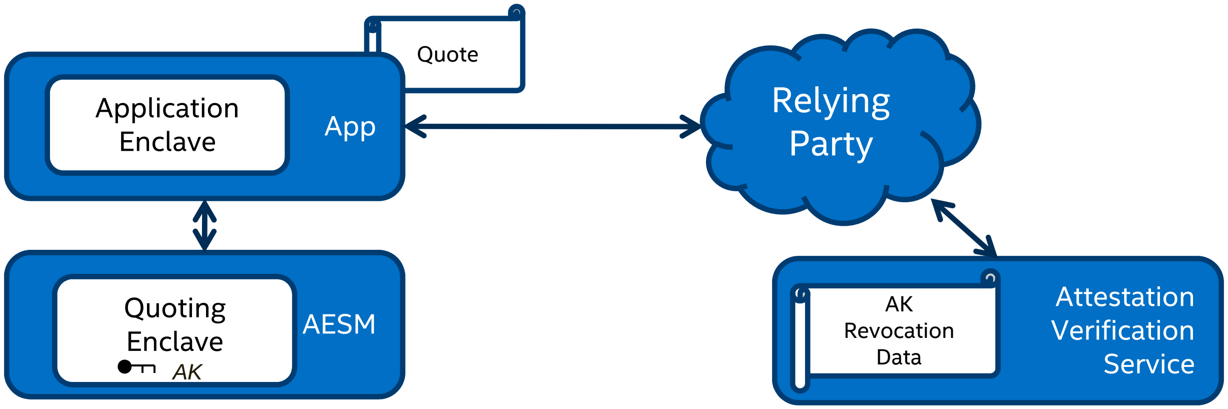
- Enclave calls EREPORT instruction to generate REPORT structure for a desired destination enclave
- REPORT structure is secured using the REPORT key of the destination enclave
- EGETKEY is used by the destination to retrieve REPORT key and then verifies structure using software.



Remote Attestation

SGX uses a quoting enclave to convert LOCAL attestations to REMOTELY verifiable assertion (QUOTE).

- Quoting Enclave (QE) locally verifies REPORT produced by Application Enclave and signs as a QUOTE.
- QE uses an asymmetric attestation key that reflects the platforms trustworthiness
- App sends Quote to the Relying Party to verify



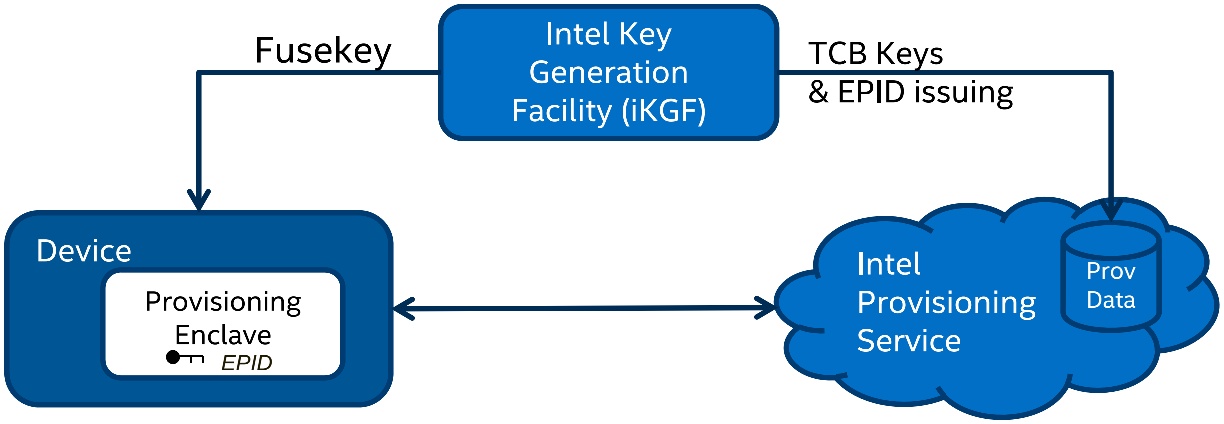
Provisioning SGX with HW-based Keys

Intel generates and fuses a unique key during manufacturing. Intel maintains a database of these keys.

Intel generates provisioning blobs for each device specific to the device TCB & Provisioning Enclave.

Protocol between Provisioning Enclave & Provisioning Service:

- Platform proves it has a key that Intel put in a real processor.
- Server certifies an Attestation Key for the platform.



Intel® EPID

Intel® Enhanced Privacy Identifier (EPID) key is used for platform identity

EPID is a Group based anonymous signature scheme.

EPID requires 3 keys:

- Master Issuing Key
- Group Public Key
- Member Private Key

Member private keys will be generated on platform.

Public keys and revocation lists to need to be issued to EPID verifiers.

Sealing

“Sealing”: Cryptographically protecting data when it is stored outside enclave.

Enclaves use EGETKEY to retrieve a persistent key that is enclave & platform specific

EGETKEY uses a combination of enclave attributes and platform unique key to generate keys

- Enclave Identity
- Enclave Sealing Authority & Product Identity

Enclave is responsible for performing the encryption with an algorithm of its choice.

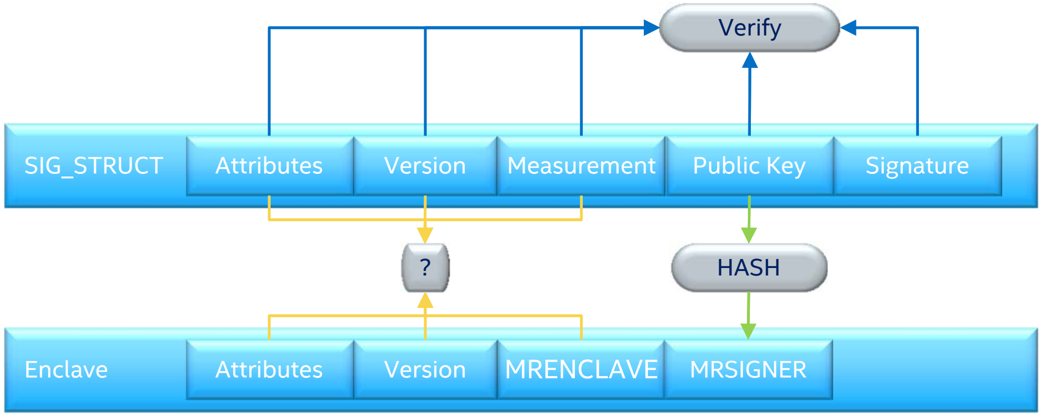
Data

Hardware

Sealing Authority

Encrypting data using keys based on hash based identities is difficult manage across software upgrades

SGX supports the notion of a Sealing Authority derived from the Enclave Certificate (SIGSTRUCT)



SGX does not check the veracity of the public key

- This is left to the relying party before they provision the enclave

SGX Security Model

Overview

During the development of Software Guard Extensions (SGX), we developed an adversary model and a set of security objectives for the architecture

We formalized the architecture and security objectives to be able to use automated tools to analyze the architecture

Adversary Model

The adversary model is broken down into software and hardware adversaries.

Software adversaries:

- Unprivileged Software
- System Software
- Startup code / System Management Mode Adversary
- Side Channel

This talk will focus on the software adversaries.

Unprivileged Software Adversary

An adversary with these capabilities is typically known as a “ring-3” attacker, whose capabilities are limited by IA-32 to those granted by the system software.

Read memory mapped into address space by system software.

Write memory mapped into address space with write privilege granted by system software.

Execute ring-3 instructions from memory mapped into address space with execute privilege granted by system software.

Execute from within an enclave.

System Software Adversary

An adversary with these capabilities has full control over the operating system, and virtual machine monitor. This adversary is capable of manipulating the Intel Instruction Set Architecture (IA-32 ISA) in any manner allowed by the ISA specification.

Control scheduling of execution

Control the mode of execution (protected, 32/64-bit compatibility, VMX host/guest)

Read all of visible memory (incl page tables, etc)

Write to all of visible memory (incl page tables, etc)

Read architectural registers (GPRs, MSRs, CRs, CSRs)

Write unlocked architectural registers (GPRs, MSRs, CRs, CSRs, including range registers, SGX owner epoch)

Execute an enclave (including a malicious enclave)

Program critical system hardware devices (memory controller, DMA engines)

Program page tables/EPT to cause faults or aliasing

Control contents of cache(indirectly with high probability)

Cause reset and control power states

Control which signed released CPU patches and other platform firmware are installed

Startup code / SMM Adversary

An adversary with these capabilities has all the capabilities of the System Software Adversary with the addition of control over initial boot code and system management mode. This adversary is capable of manipulating the IA-32 ISA in any manner allowed by the ISA specification.

Has full control over system state during the startup from S3, S4 and S5.

Modify the contents of SMM RAM in a way invisible to OS or VMX host.

Create virtual devices depending on intercept controls.

Intercept some limited set of software actions and emulate behavior following intercept.

Make persistent modifications to the state of system firmware on platforms not implementing stronger protections such as BIOS Guard.

Modify protected MSR’s/CSR’s that are open only to SMM

Software Side-channel Adversary

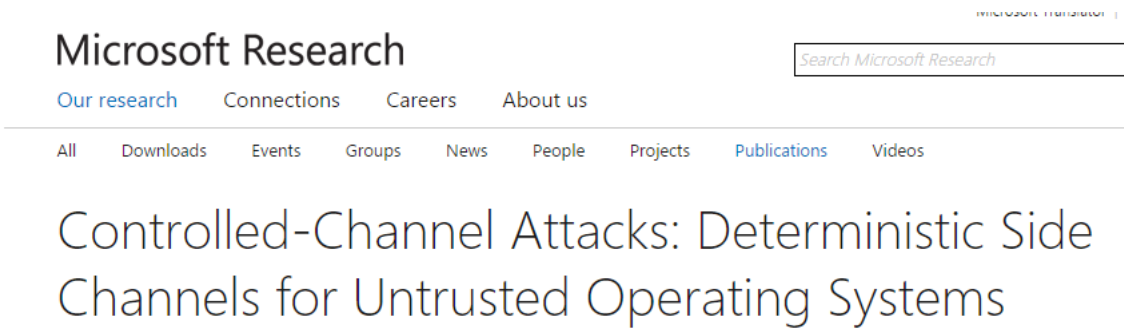
An adversary with these capabilities is able to gather statistics from the CPU regarding execution and may be able to use them to deduce characteristics of the software being executed (side-channel analysis).

Gather power statistics

Gather performance statistics including platform cache misses

Gather branch statistics via timing

Gather information on pages accessed via page tables



Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems

SGX does not defend against this adversary

Definitions

TERM	DEFINITION
Enclave Binary	The initial code and data which is loaded by the system software and measured by the hardware.
Enclave Class	The attested identity of an enclave.
Enclave Instance	An enclave binary which has been initialized and verified by the hardware to match a provided enclave class identity.
System Software	Operating system (OS), virtual machine monitor (VMM), system management mode (SMM), BIOS, or other privileged software which controls the operation of the system.

Security Objectives: Execution Environment

Detection of an integrity violation of an enclave instance computation from software attacks by an unprivileged, system software attacker, or system startup/SMM and prevent access to tampered code/data upon detection.

Provide confidentiality of code/data of an enclave instance from software attacks by an unprivileged, system software attacker, or system startup/SMM.

Provide isolation between all enclave instances.

Security Objectives: Execution Environment

Unprivileged software attacker cannot bypass the existing IA-32 access control mechanisms.

- An enclave continues to abide by the read/write/execute access control policy set by system software.
- No extra privileges beyond what system software grants.

Prevent replay of an enclave instance from software attacks by an unprivileged, system software attacker, or system startup/SMM.

Security Objectives: Remote Provisioning

Allow an enclave instance to request an assertion from the platform of the enclave's identity.

It is computationally infeasible for an unprivileged software, system software, simple hardware, or skilled hardware attacker to spoof the assertion.

Allow an enclave instance to verify the SGX-provided assertion originated from the same platform.

Provide a mechanism that allows any entity to verify an assertion from an enclave instance.

Security Objectives: Remote Provisioning

Allows an enclave instance to obtain keys that are bound to the platform and the enclave class identity or a subset of the enclave class identity.

- These keys can be used for data sealing, provisioning, ...

Prevents access to other enclave class keys by an unprivileged software, system software, simple hardware, or skilled hardware attacker.

- Including other enclaves of different class identities

Security Objectives: Key Management

It is computationally infeasible for a software adversary to use a compromised TCB to access the secrets of a larger (newer) TCB revision.

It is computationally infeasible for a software adversary to enable more than one certified platform key per TCB SVN.

Side Channel Detection Tool

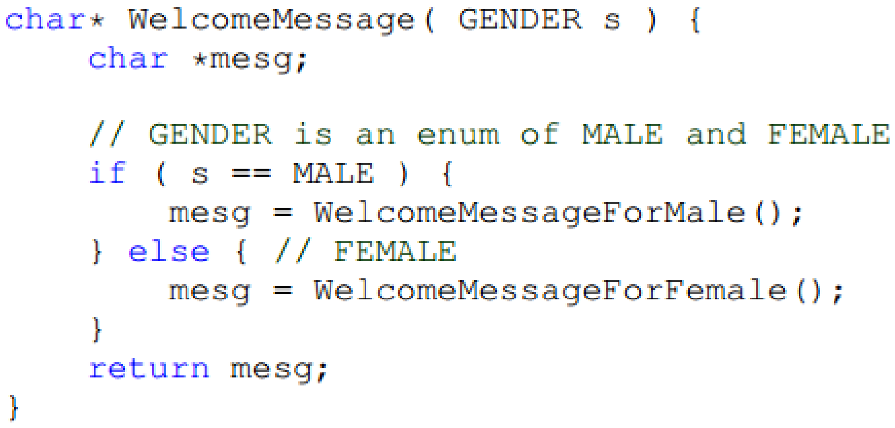
A new type of side-channel attacks

Goal

- Recognize input-dependent control transfers and data accesses

How

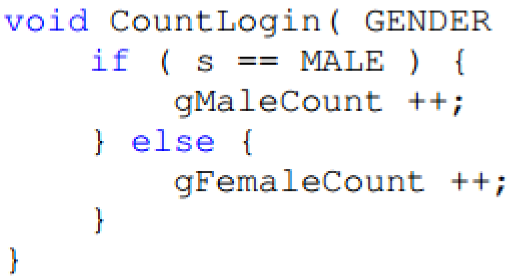
- Force page-faults on each new page access
- Compare with known page-fault patterns to infer input values



```
char* WelcomeMessage( GENDER s ) {
    char *mesg;

    // GENDER is an enum of MALE and FEMALE
    if ( s == MALE ) {
        mesg = WelcomeMessageForMale();
    } else { // FEMALE
        mesg = WelcomeMessageForFemale();
    }
    return mesg;
}
```

Fig. 1: Example function with input-dependent control transfer.



```
void CountLogin( GENDER s ) {
    if ( s == MALE ) {
        gMaleCount ++;
    } else {
        gFemaleCount ++;
    }
}
```

Fig. 2: Example function with input-dependent data access.

Control-channel Attacks: Deterministic Side Channels for Untrusted Operating Systems; Xu et al, 36th IEEE Symposium on Security and Privacy, 2015

Detecting potential control-channel attack locations

1. Static techniques

- Results valid for all possible executions
- Compiler modifications required

2. Dynamic techniques

- Easier to implement (thanks to Pin!)
- Results valid only for a specific execution
  - Will need multiple executions to improve coverage

Dynamic techniques

To automatically find control transfers and data accesses that are input dependent

Challenges:

1. Overhead (need per instruction def/use info)
2. Coverage : Only valid for a specific execution

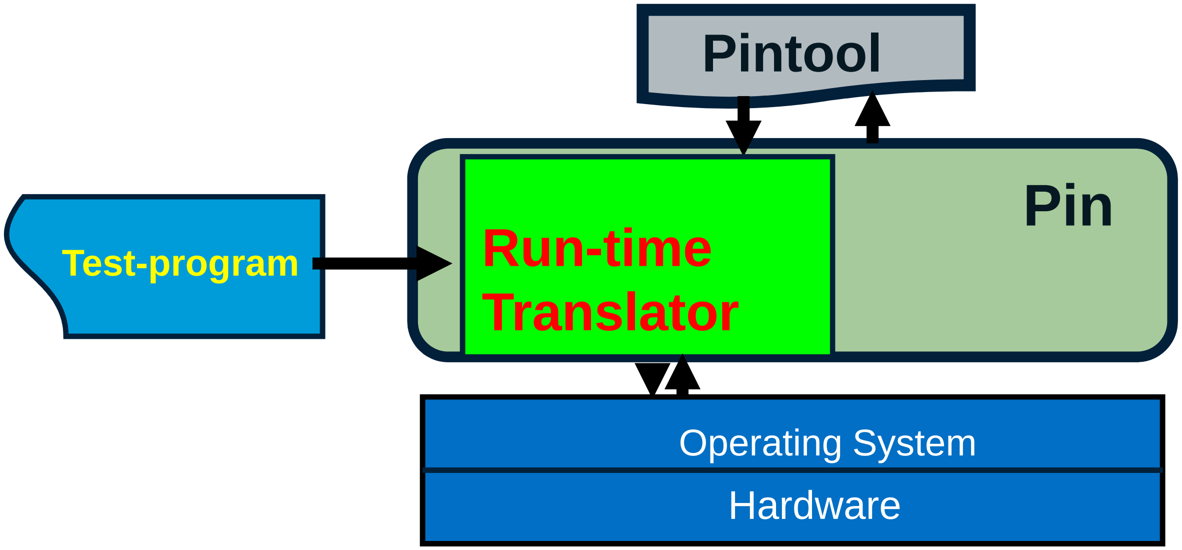
Candidates:

1. Dynamic Taint Analysis tool : This presentation
2. Forward dynamic slicing : Later this summer

Pin: A Tool for Writing Program Analysis Tools

Pin: A Dynamic Instrumentation Framework from Intel
http://www.pintool.org
> 30,000 downloads in 10 years

Pin: A Tool for Writing Program Analysis Tools

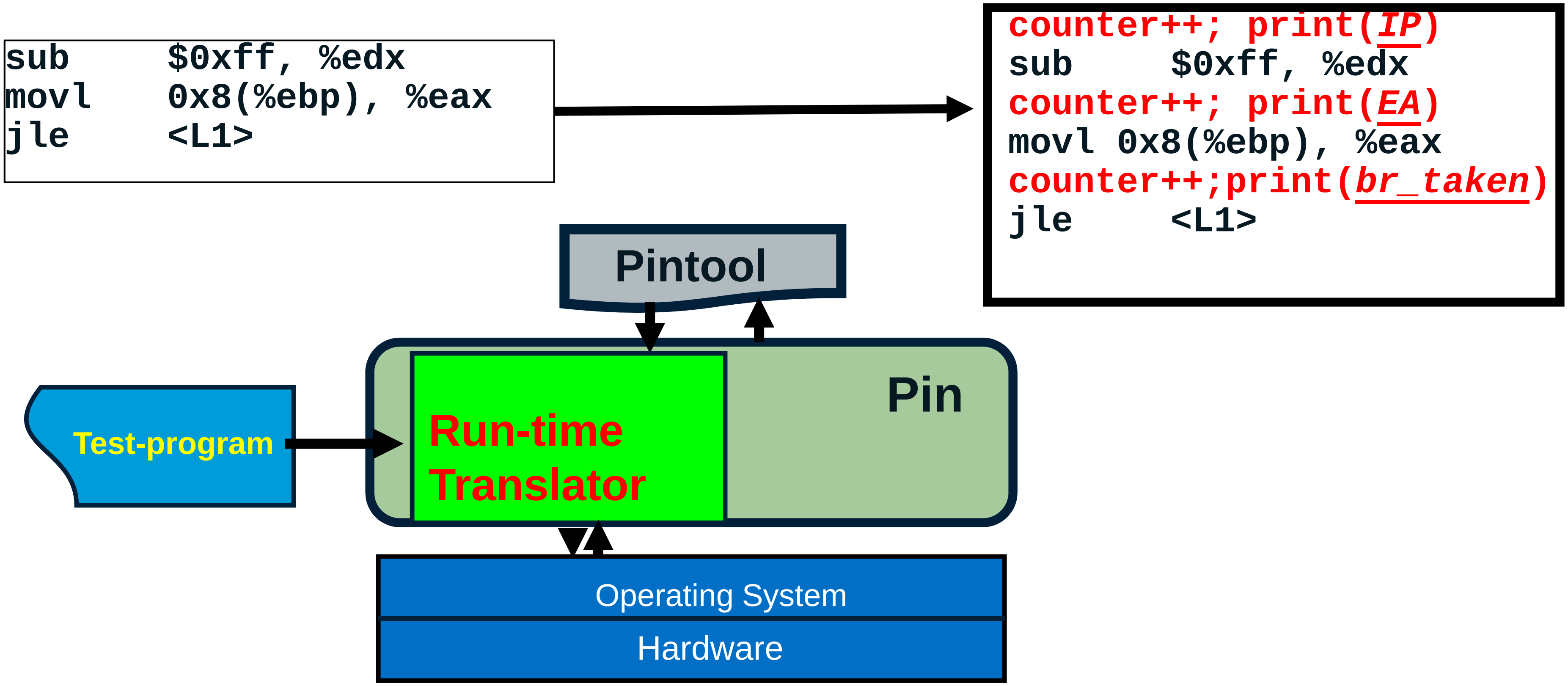


Pin: A Dynamic Instrumentation Framework from Intel

http://www.pintool.org

> 30,000 downloads in 10 years

Pin: A Tool for Writing Program Analysis Tools



```
sub     $0xff, %edx
movl    0x8(%ebp), %eax
jle     <L1>
```

```
counter++; print(IP)
sub     $0xff, %edx
counter++; print(EA)
movl 0x8(%ebp), %eax
counter++;print(br_taken)
jle     <L1>
```

Pintool

Pin

Test-program

Run-time Translator

Operating System

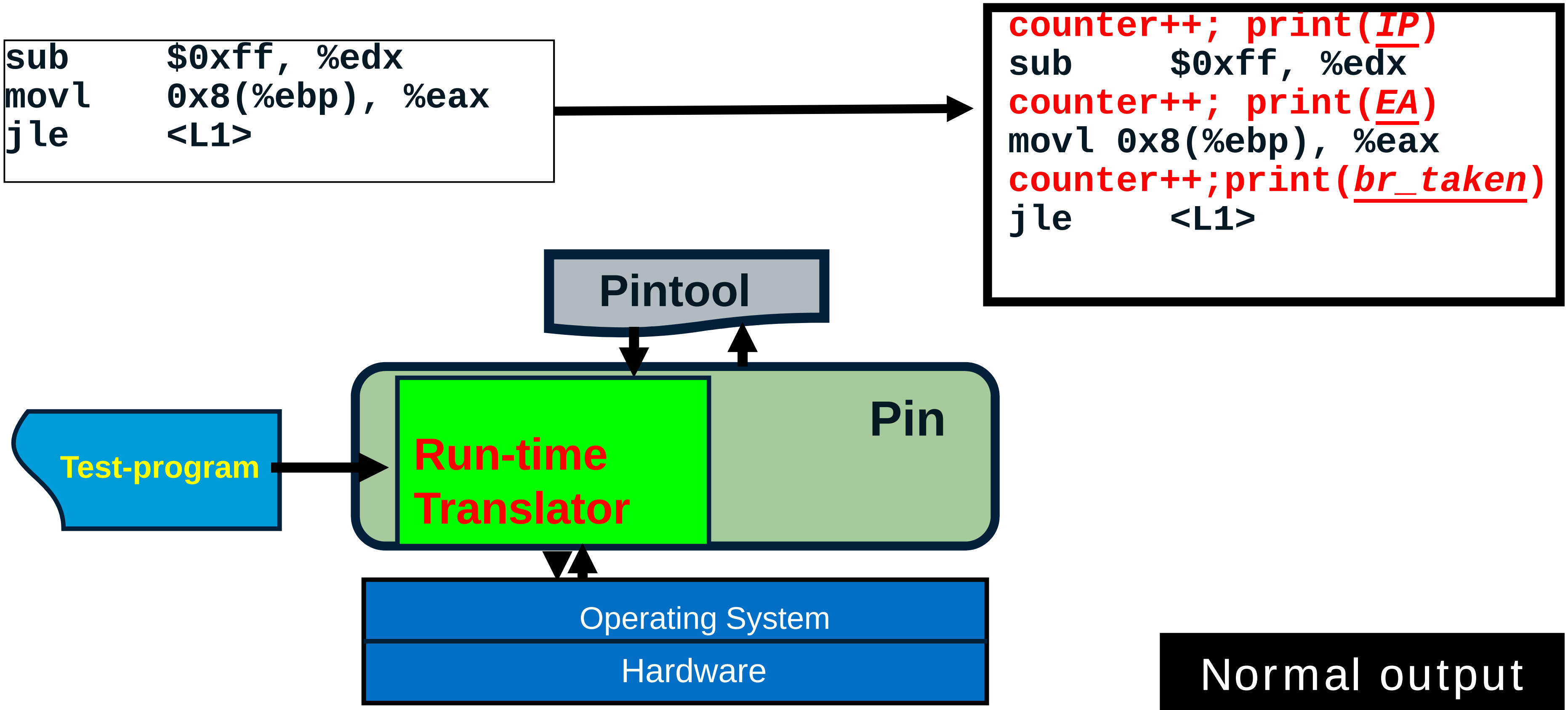
Hardware

Pin: A Dynamic Instrumentation Framework from Intel

http://www.pintool.org

> 30,000 downloads in 10 years

Pin: A Tool for Writing Program Analysis Tools



```
sub     $0xff, %edx
movl    0x8(%ebp), %eax
jle     <L1>
```

```
counter++; print(IP)
sub     $0xff, %edx
counter++; print(EA)
movl 0x8(%ebp), %eax
counter++;print(br_taken)
jle     <L1>
```

```
$ pin –t pintool –– test-program
```

Normal output
+ *Analysis output*

Pin: A Dynamic Instrumentation Framework from Intel
http://www.pintool.org
> 30,000 downloads in 10 years

Why Pin?

Easy to use: *"if you can run it you can Pin it"*

- No special hardware or operating system needed
- No program source code needed
- No re-linking required

Multi-platform:

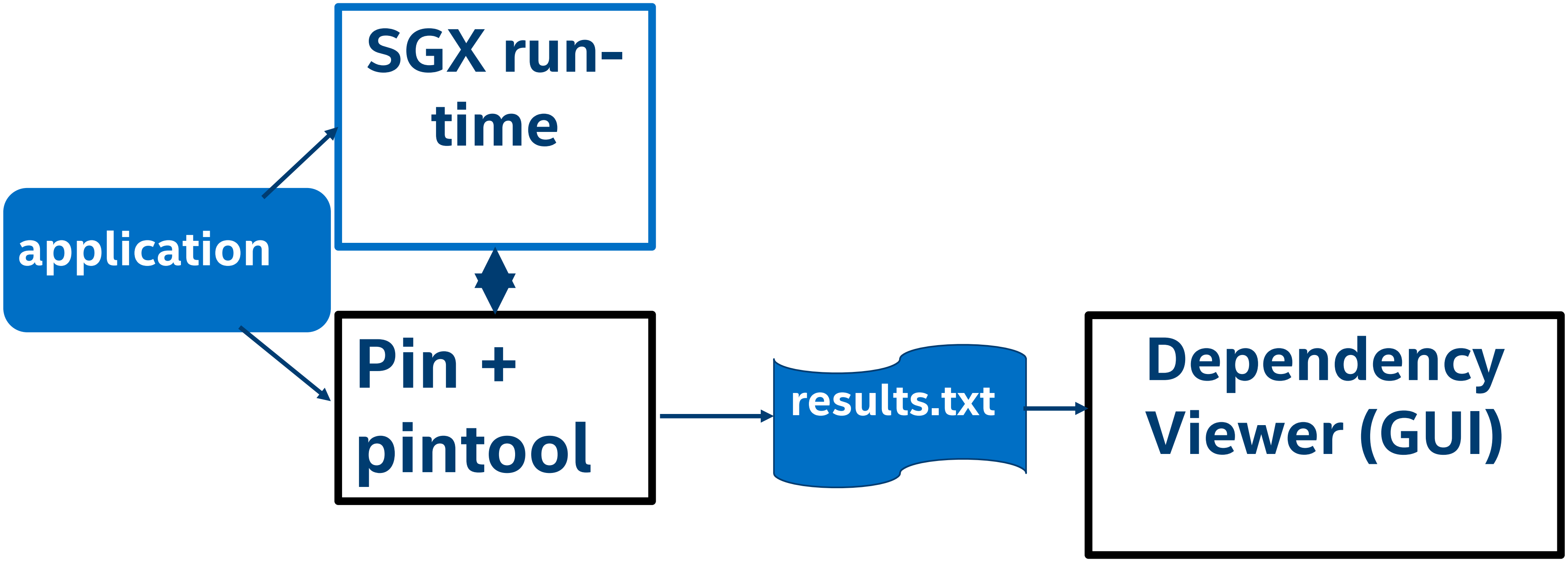
- Supports {IA-32, Intel64} {Linux, Windows, Android, MacOS}

Robust:

- Instruments real-life applications
  - Database, search engines, web browsers, …

Supported by Pin team at Intel

Taint Analysis Tool (TAT) [SGX : simulation mode only]

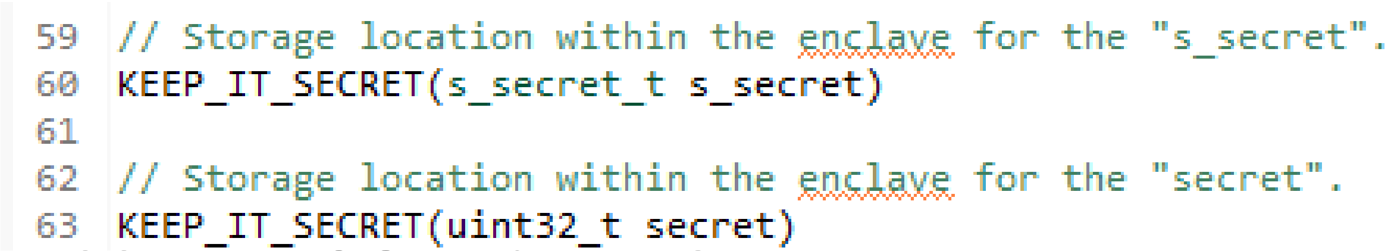


Developers: Michael Gorin, Juan Del Cuvillo, Harish Patil, David Wootton

Longer term goal: Integrated GUI + PinTool

TAT : Developer marks ‘sensitive input’s

Sensitive data identified with a macro



```
// Storage location within the enclave for the "s_secret".
KEEP_IT_SECRET(s_secret_t s_secret)

// Storage location within the enclave for the "secret".
KEEP_IT_SECRET(uint32_t secret)
```

Pi

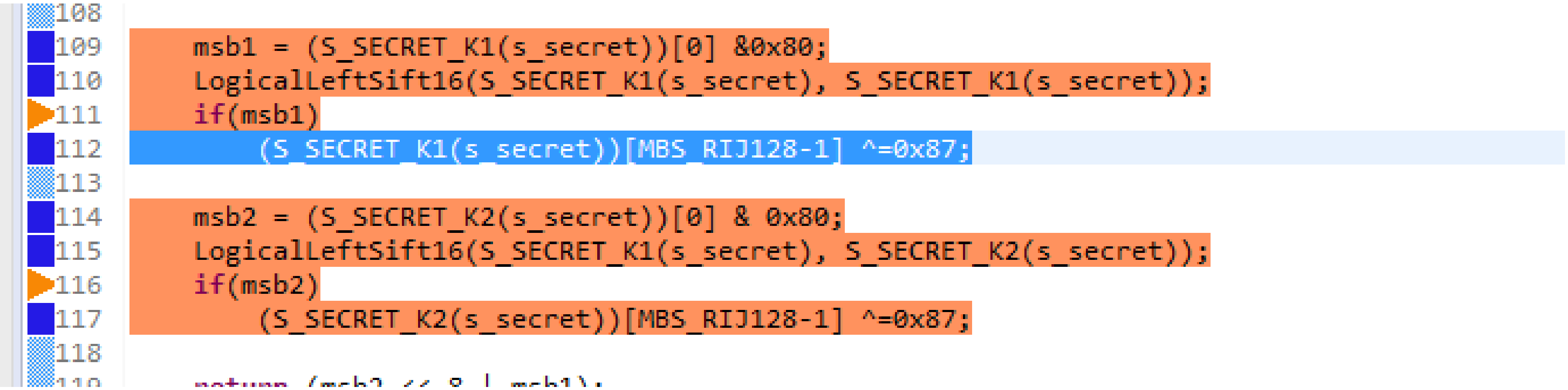
- Number, address, size, names of ‘sensitive data’ variables (globals currently)

TAT Pintool:

- Tracks data flow of sensitive data across instructions

Screenshots: Dependency Viewer

Summary view: All statements affected



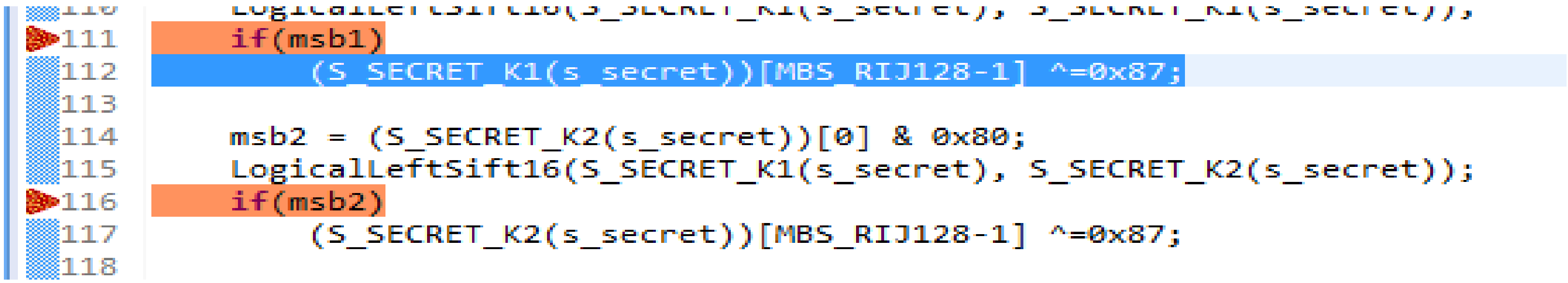
```

    msb1 = (S_SECRET_K1(s_secret))[0] &0x80;
    LogicalLeftSift16(S_SECRET_K1(s_secret), S_SECRET_K1(s_secret));
    if(msb1)
        (S_SECRET_K1(s_secret))[MBS_RIJ128-1] ^=0x87;

    msb2 = (S_SECRET_K2(s_secret))[0] & 0x80;
    LogicalLeftSift16(S_SECRET_K1(s_secret), S_SECRET_K2(s_secret));
    if(msb2)
        (S_SECRET_K2(s_secret))[MBS_RIJ128-1] ^=0x87;

    return (msb2 << 8 | msb1);
```

Branch View: Secret used for control-flow



```
    LogicalLeftSift16(S_SECRET_K1(s_secret), S_SECRET_K1(s_secret));
    if(msb1)
        (S_SECRET_K1(s_secret))[MBS_RIJ128-1] ^=0x87;

    msb2 = (S_SECRET_K2(s_secret))[0] & 0x80;
    LogicalLeftSift16(S_SECRET_K1(s_secret), S_SECRET_K2(s_secret));
    if(msb2)
        (S_SECRET_K2(s_secret))[MBS_RIJ128-1] ^=0x87;

```

Summary

Prototype tool for finding input-dependent control-transfers and data access

- **Dynamic** (Pin based)
  - Valid for only a given execution
  - analyze only in Enclave → efficient
- **Next steps**:
  - Allow local secret tracking
  - Explore forward-slicing (Linux)

Formal Methods for Security Validation

Introduction

Formal verification is a collection of techniques that exhaustively test a design for a particular property

- For example, model checking or theorem proving

Formal techniques are most valuable when applied to large systems where testing cannot provide adequate coverage

- A model of the system captures the relevant information about the system
- A collection of models may be used to evaluate a variety of system properties

During the development of Software Guard Extensions (SGX), the team employed formal verification to validate the security objectives of the SGX design

- Identified critical architecture bugs early in the design process
- Evaluated security impact of design changes over time

SGX Verification Goals

Sanity check SGX behavior

- Instructions maintain well-formedness of the EPCM and enclave state
- Common sequences of instructions produce expected results

Demonstrate that the SGX architecture provides strong separation between an enclave and its environment in a single threaded setting

Demonstrate that multiple threads simultaneously executing SGX instructions cannot place the hardware in an inconsistent state

Talk Plan

Sequential Verification in DVF

- High-level model of SGX architecture used for sanity checking and separation property validation

Verifying Concurrency Correctness in iPave

- Detailed model of SGX hardware state used for finding unsafe races between instructions

Automatic Linearizability Analysis with Accordion

- New tool that reduces the manual effort for concurrency correctness checking

Sequential Verification in DVF

Tool Overview - DVF

Semi-automatic verification tool developed at Intel

- See http://smt2012.loria.fr/paper2.pdf for details

Validator creates a model of the system in the DVF specification language and defines the desired properties of the model

DVF attempts to automatically prove each property by invoking one of several back-end verification tools (e.g., Yices SMT solver)

Complex properties may not be automatically verifiable, in which case the validator breaks the property down into a series of smaller proof steps that can be automatically verified by the tool

Anatomy of a DVF Model

DVF models describe state transition systems

Global variables define the system state

- SGX internal state
- Memory, including Enclave Page Cache (EPC)
- Logical processors and thread-local state

State transition functions describe the effect of system execution on the system state

- Enclave instructions
- Memory read/write
- Thread execution

Formal properties describe the validation goals

- A property is valid if it holds in *every* system state

Example: EADD Specification in DVF

```
transition eadd_reg(linear_address a, linear_address s, linear_address src)
  require(mapped(a))
  require(mapped(s))
  require(mapped(src))
  require(maps_to_uninitialized_secs(s))
  require(in_secs_els(lookup_page(s),from_linear(a)))
  require(maps_to_free_epc_address(a))
  require(!in_enclave_mode(curr_lp))
  require(!is_epc_address(lookup_page(src)))
  {
    epcm[lookup_page(a)] := {
      valid          = true;
      pt             = pt_reg;
      enclavesecs    = lookup_page(s);
      enclaveaddress = from_linear(a);
    };

     mem[pa] := REG(contents(src));
  }
```

Example: EADD Specification in DVF

```
transition eadd_reg(linear_address a, linear_address s, linear_address src)
  require(mapped(a))
  require(mapped(s))
  require(mapped(src))
  require(maps_to_uninitialized_secs(s))
  require(in_secs_els(lookup_page(s),from_linear(a)))
  require(maps_to_free_epc_address(a))
  require(!in_enclave_mode(curr_lp))
  require(!is_epc_address(lookup_page(src)))
  {
    epcm[lookup_page(a)] := {
      valid          = true;
      pt             = pt_reg;
      enclavesecs    = lookup_page(s);
      enclaveaddress = from_linear(a);
    };

     mem[pa] := REG(contents(src));
  }
```

Signature:

- **Defines a state transition function**
- **Transitions are evaluated using arbitrary values for parameters**
- **No hidden corner cases**

Example: EADD Specification in DVF

```
transition eadd_reg(linear_address a, linear_address s, linear_address src)
  require(mapped(a))
  require(mapped(s))
  require(mapped(src))
  require(maps_to_uninitialized_secs(s))
  require(in_secs_els(lookup_page(s),from_linear(a)))
  require(maps_to_free_epc_address(a))
  require(!in_enclave_mode(curr_lp))
  require(!is_epc_address(lookup_page(src)))
  {
    epcm[lookup_page(a)] := {
      valid          = true;
      pt             = pt_reg;
      enclavesecs    = lookup_page(s);
      enclaveaddress = from_linear(a);
    };

     mem[pa] := REG(contents(src));
  }
```

Preconditions:

- **Define restrictions on when state transition is valid**
- **Should correspond to explicit checks or guarantees in the HW**

Example: EADD Specification in DVF

```
transition eadd_reg(linear_address a, linear_address s, linear_address src)
  require(mapped(a))
  require(mapped(s))
  require(mapped(src))
  require(maps_to_uninitialized_secs(s))
  require(in_secs_els(lookup_page(s),from_linear(a)))
  require(maps_to_free_epc_address(a))
  require(!in_enclave_mode(curr_lp))
  require(!is_epc_address(lookup_page(src)))
  {
    epcm[lookup_page(a)] := {
      valid          = true;
      pt             = pt_reg;
      enclavesecs    = lookup_page(s);
      enclaveaddress = from_linear(a);
    };

     mem[pa] := REG(contents(src));
  }
```

Specification Body:

- **Describes changes instruction makes to system state**
- **Implicitly relies on a provided input state**
- **Bugs are found by searching for input states/parameters that violate the desired system invariants**

Properties Verified

SGX invariants

- SGX instructions maintain well-formedness of the EPCM and enclave state

Enclave Confidentiality

- Enclave data cannot be read outside the enclave
- Secrets do not survive enclave teardown

Enclave Data Integrity

- Enclave data cannot be modified or manipulated outside the enclave

Lifecycle Properties

- SGX instructions update enclave and page state in an expected fashion
- Example: Calling EADD on an appropriately mapped invalid page produces a valid page of the requested type (REGULAR or TCS)

Example: Formalizing Confidentiality

Data contained in an enclave page may only be read by that enclave

```
goal c0 =
  {let pa = page_table[a] in
     is_epc_address(pa)
     && (!valid(pa)
         || !thread_in_enclave(curr_lp,enclaveowner(pa))
  }
  read(a)
  {result = abort_data}
```

A page has not been allocated at ***pa*** in the enclave to which ***pa*** belongs

OR the executing thread is not currently executing in the enclave to which ***pa***

A memory read will not produce any data

Example: Formalizing Confidentiality

Data contained in an enclave page may only be read by that enclave

```
goal c0 =
  {let pa = page_table[a] in
     is_epc_address(pa)
     && (!valid(pa)
         || !thread_in_enclave(curr_lp,enclaveowner(pa))
  }
  read(a)
  {result = abort_data}
```

This property is just one of several that comprise our proof that SGX does not leak enclave secrets

DVF Modeling Results

Employed during SGX design process to formalize the security properties of the architecture

- Found bugs
- Increased assurance in architecture design

≈ 2200 lines of code (800 model, 1400 proof)

Property breakdown:

- 16 EPCM and enclave well-formedness invariants
- 68 security and lifecycle properties that describe the behavior of instructions in particular states

Verifying Concurrency Correctness in iPave

Concurrency in SGX

Instructions may be executing concurrently on different processors

- Validators cannot make any assumptions about how instructions will be used

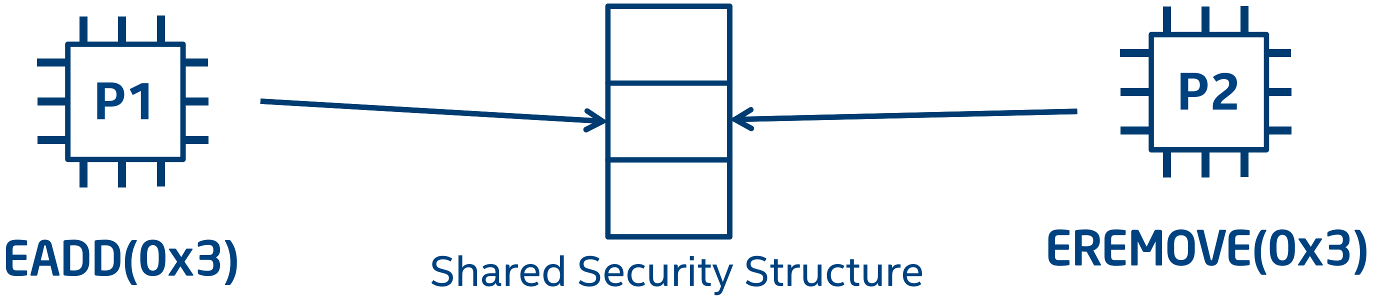
A shared hardware data structure stores enclave security information

- Instructions simultaneously access the shared data structure

Hardware locks are used to protect shared state access

- Synchronization is kept to a minimum for performance and flexibility

Non-standard protocols make existing lock analyses inapplicable



Concurrency Correctness Criterion: Linearizability

Classic approach for reasoning about concurrent accesses to a shared data structure (Herlihy and Wing, 1990)

A system is *linearizable* if:

- Each operation appears to take effect atomically at some moment in time between its invocation and response, called its *linearization point*

Local property that does not depend on run-time scheduling

- Each execution of a linearizable system is equivalent to a known sequential trace based on the linearization points of the operations

If a system is linearizable, we may reason about the correctness of the system with sequential techniques

- Ensures that our security proofs in the sequential setting are sound

Proving Linearizability

Proof by construction:

- Identify the linearization point of each concurrent instruction
- Assume that each operation takes effect atomically at its linearization point
- Demonstrate that the observable effects of the actual system match the observable effects of the linearized system

Demonstration technique:

- Create a model of the system that captures the real system's interactions with the concurrent data structure of interest
- At each linearization point, compare the portion of the real system state accessed by the operation to the same portion of the linearized system state
- Difference in state = concurrency bug!
- Flexible method that can be evaluated using formal methods or testing

Tool Overview - iPave

Graphical specification language and automatic verification tool developed at Intel

Validator draws a model of the system in Visio and defines the system invariants in a separate text file

Model checker explores the reachable states of the model, checking the system invariants before each state transition

- Exploration is bounded to a certain depth

Complex systems may take too long to return results, in which case the validator optimizes the model to contain less system details

Anatomy of an iPave Model

Global variables define the system state

- A single concrete EPCM entry and an abstract representation of “other” entries
- Address translation cache
- Other SGX internal state, such as logical processor state

Boxes represent intermediate states

- Potential interleaving point with other threads

State transition arrows describe constraints on system execution and updates to the system state

- Enclave instructions
- Memory read/write, caching

Properties are expressed as invariants or per-state assertions

Properties Verified

Linearzability assertions

- Any field in the EPCM accessed by the instruction is not changed by another thread between the time of the access and the linearization point
- Checked at the linearization point of each instruction (after last write to EPCM state but before lock release)

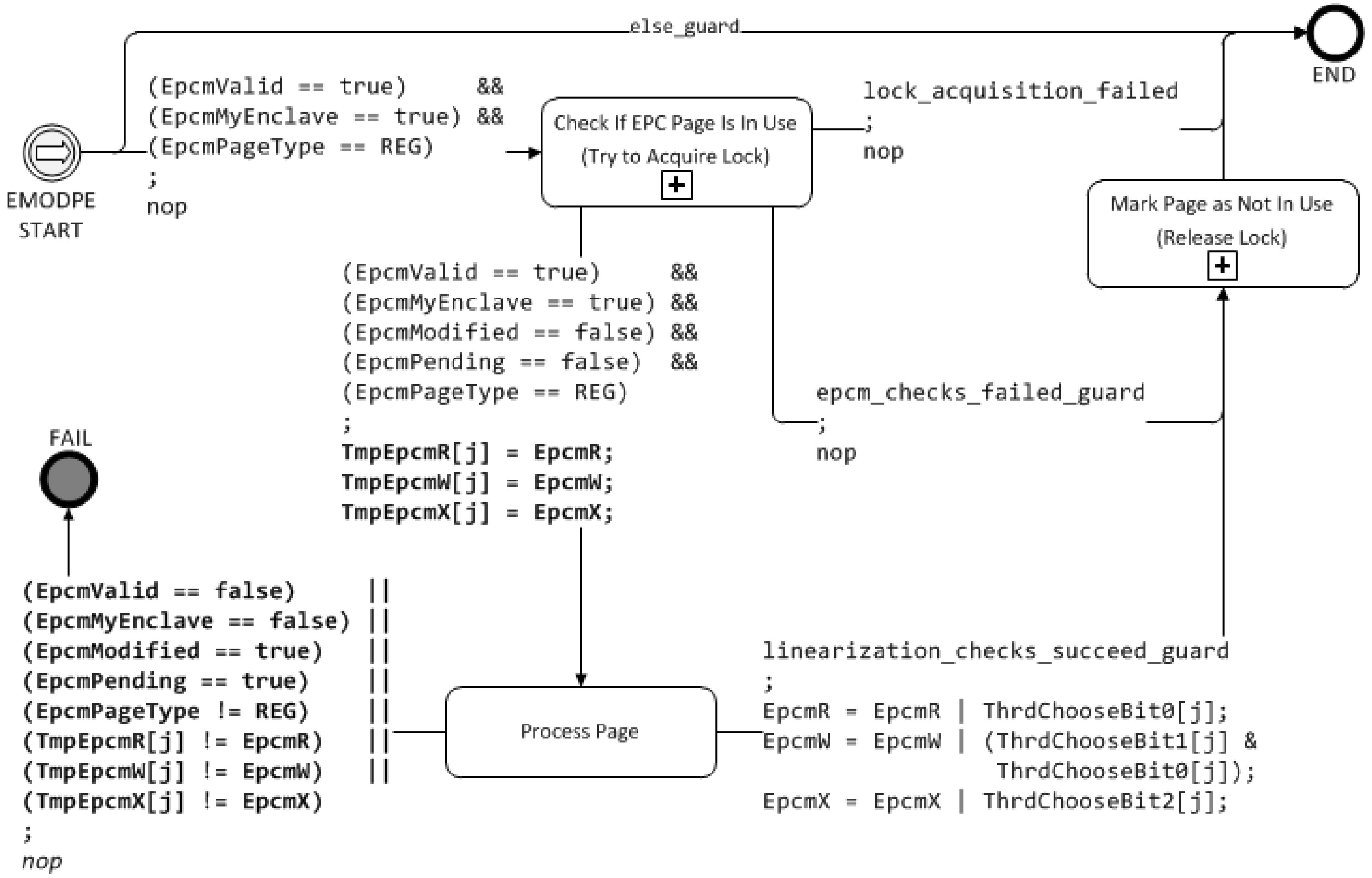
SGX invariants

- Checked in every state
- identify bugs that were not caught by the more abstract DVF model
- Example: A PENDING page is never cached

NB: iPave performs a bounded search

- Result of iPave run is not a full proof
- Use of explicit state model checker (see next section) can provide greater confidence in the result with a performance penalty

Example – EMODPE Specification



```
else_guard
```

```
(EpcmValid == true)      &&
(EpcmMyEnclave == true)  &&
(EpcmPageType == REG)
;
nop
```

```
lock_acquisition_failed
;
nop
```

```
(EpcmValid == true)      &&
(EpcmMyEnclave == true)  &&
(EpcmModified == false)  &&
(EpcmPending == false)   &&
(EpcmPageType == REG)
;
TmpEpcmR[j] = EpcmR;
TmpEpcmW[j] = EpcmW;
TmpEpcmX[j] = EpcmX;
```

```
epcm_checks_failed_guard
;
nop
```

```
(EpcmValid == false)      ||
(EpcmMyEnclave == false)  ||
(EpcmModified == true)    ||
(EpcmPending == true)     ||
(EpcmPageType != REG)     ||
(TmpEpcmR[j] != EpcmR)    ||
(TmpEpcmW[j] != EpcmW)    ||
(TmpEpcmX[j] != EpcmX)
;
nop
```

```
linearization_checks_succeed_guard
;
EpcmR = EpcmR | ThrdChooseBit0[j];
EpcmW = EpcmW | (ThrdChooseBit1[j] &
                 ThrdChooseBit0[j]);
EpcmX = EpcmX | ThrdChooseBit2[j];
```

iPave Modeling Results

Formal verification fills a gap in the current validation methodology

- State space of multi-threaded execution too large for exhaustive testing
- Concurrency bugs are often too subtle for detection via manual review

iPave model identified critical security bugs in rooted in the concurrent interactions between SGX instructions

- High quality bugs not found by previous testing or code review

Formal verification provides confidence that we are not overlooking bugs in the architecture design

Automatic Linearizability Analysis with Accordion

Tool Overview - Accordion

An ***embedded domain specific language***

- Designed for describing instruction flows
- Visually similar to existing SGX specification
- Reusable for future instruction set extensions

A ***compiler***

- Targets multiple model checkers
- Automates linearizability analysis
- Optimizes generated model

For details see our CAV 2015 paper:

- Verifying Linearizability of Intel Software Guard Extensions

Summary

Formal verification is an integral part of the SGX design process

- Identified security-critical architecture bugs
- Gives confidence in the correctness of the final architecture

A variety of tools are employed to tackle SGX validation goals

- Proof in DVF
- Model checking
- Design exploration with Accordion

Application of formal verification techniques to SGX began as a research endeavor, but is now performed by a core member of the validation team

Ongoing research tackles scalability and future SGX functionality

Questions?

For more information see our paper at CAV 2015:

“Verifying Linearizability of Intel Software Guard Extensions”

Memory Encryption Engine Architecture and Security Properties

Outline

Memory protection: problem statement and background

Design challenges

The abstract MEE model

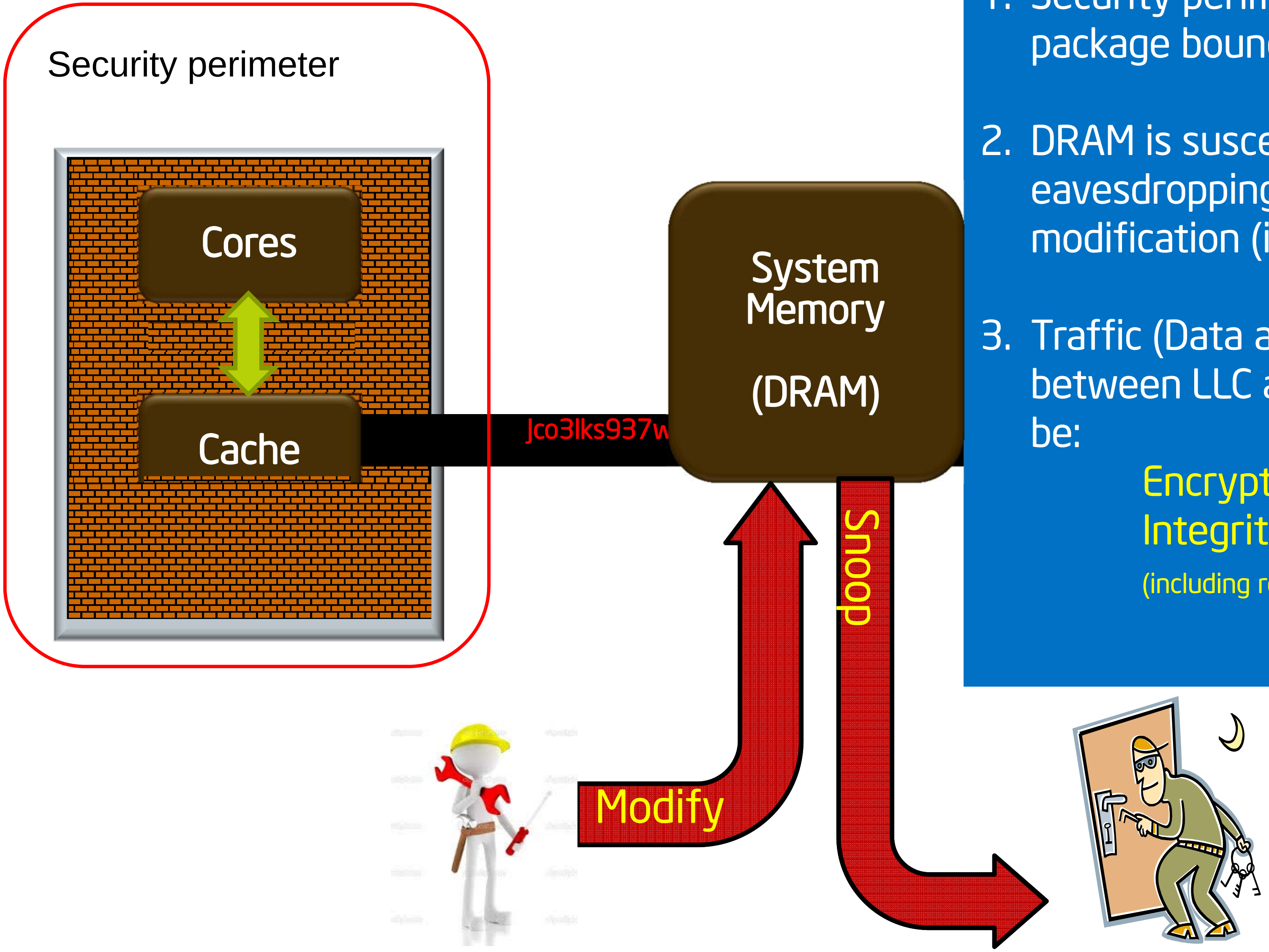
The MEE cryptographic schemes (confidentiality and integrity)

Establishing cryptographic security bounds

The MEE operation

Putting the cryptographic bounds to the test

The protected memory problem



1. Security perimeter is the CPU package boundary
2. DRAM is susceptible to passive eavesdropping and malicious modification (including replay)
3. Traffic (Data and Code) between LLC and DRAM must be:

   Encrypted,

   Integrity checked

   (including replay protected)

Memory Encryption Engine in a nutshell

Intel® SGX is supported by a the Memory Encryption Engine (MEE)

MEE Operates as an extension of the Memory Controller (MC)

MEE Protects confidentiality & integrity part of the system memory

- Covers a selected range of memory addresses (MEE region)
- Uses a set of secret keys generated uniformly, at random, during boot
- Most of the MEE region resides on the DRAM; a small portion resides on die

MEE is based on a careful combination of cryptographic primitives, operating over an efficient data structure.

Designed to achieve the cryptographic properties, while satisfying very strict engineering constraints.

Build

How the MEE works – in a nutshell

Core issues a transaction

- e.g., WRITE

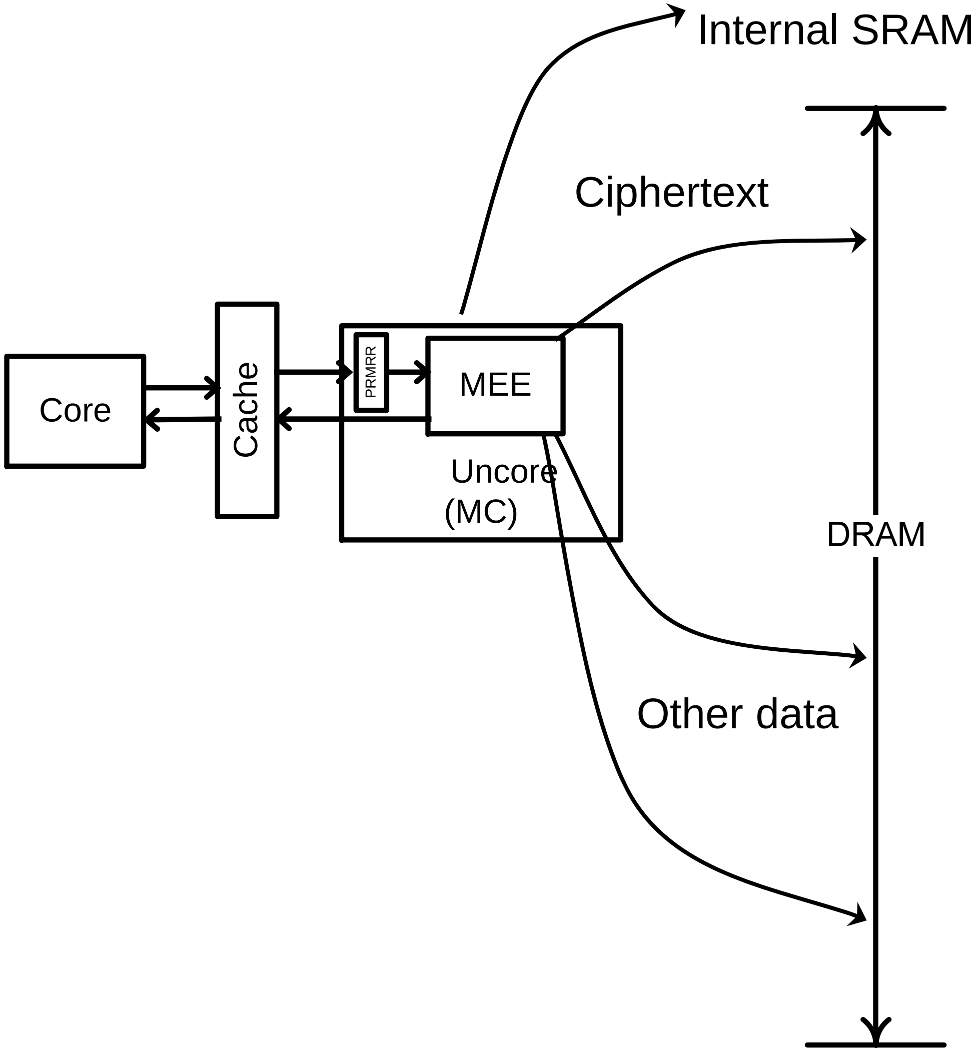
Transaction misses caches and forwarded to Memory Controller

MC detects address belongs to MEE region, and routes transaction to MEE

Crypto processing…

MEE initiates additional memory accesses to obtain (or write to) necessary data from DRAM

- Produces plaintext (ciphertext)
- Computes authentication tag
- (uses/updates internal data)
- writes ciphertext + added data



The Design Challenges

Need data confidentiality and integrity? Hold everything internally

A. Very (very) expensive – not a practical option

Need data confidentiality and integrity? Standard encryption/ authentication

A. Standards are not optimal for this problem

Some of the requirements:

- Hardware efficiency (small area)
- Parallelizability between (and among) authentication and encryption
- Short MAC tags (< 64b); at least Truncatable MACS

So what does work?

A tailored AES CTR encryption

- Special Counter Block structure

A tailored MAC algorithm

- Carter-Wegman MAC over a multilinear universal hash function + truncation

A special data-structure to accommodate and compact MEE data

- Minimizing the required internal storage

Our crypto design approach

Assume an adversary with “super capabilities”

- Beyond what’s really practical

Derive the desired cryptographic properties against such an adversary

Convert the MEE problem to a cryptographic setting

Prove “information theoretic” bounds on the encryption and authentication

- Based on standard assumptions on AES
  - AES is a good Pseudo Random Permutation (PRP) of $\{0, 1\}^{128}$

Test attack feasibility under the platform’s physical limitation

Some basic MEE setup and policy

Separate keys for encryption and integrity

- Separate passive eavesdropping from active forgery

Drop-and-lock policy: Response to a MAC tag mismatch (in READ/WRITE)

  - The MEE emits a failure signal
  - Drops the transaction (i.e., no data is sent to the LLC)
  - Locks the MC (i.e., no further transactions are serviced).
  - Eventually causes the system to halt; reset is required to continue operations
  - A new boot cycle uses new keys.

- An adversary is allowed only one failed forgery attempt per key set.

System general setup:

- Read/Write operations are always at the granularity of 512 bits Cache Line (CL)

The super power adversary

- Idealized eavesdropper and an idealized active forger.

Adversary has physical access to the platform and can do (at any time):

- **Write** (using crafted software) any chosen plaintext CL to any legit address
- **Store** (off-chip) all the snapshots of the DRAM contents
  - Starting from the system initialization time
- **Modify** any number of chosen bits on the DRAM
- **Cause** (using crafted software) any CL, of any code running on the system, to be evicted from the LLC to the DRAM, or be pulled from DRAM back to LLC.
- **Run** any combination of application code and attack code, and control the OS.

- Can see the full sequence of changes on the DRAM, at the CL granularity

But cannot read/modify on-die values (MEE keys and SRAM counters).

Strategy of the (super power) adversary

The adversary has some budget of ciphertext / MAC tags samples

Observes ciphertext / tags samples with "idealized" capabilities

- Assume every observed ciphertext comes from a *chosen* plaintext
- Assume every observed MAC tag comes from a *chosen* message

Then – using the collected information:

- Tries to learn something on the encrypted data (from victim applications)
- Attempts a forgery (and can immediately sees the result)
  - Drop-and-lock policy → only one failing forgery attempt per set of keys

What is the probability that the adversary can win against the MEE?

MEE desired properties

Data Confidentiality:

- Collections of memory images of DATA written to the DRAM (into different addresses and points in time) cannot be distinguished from random data.

Integrity:

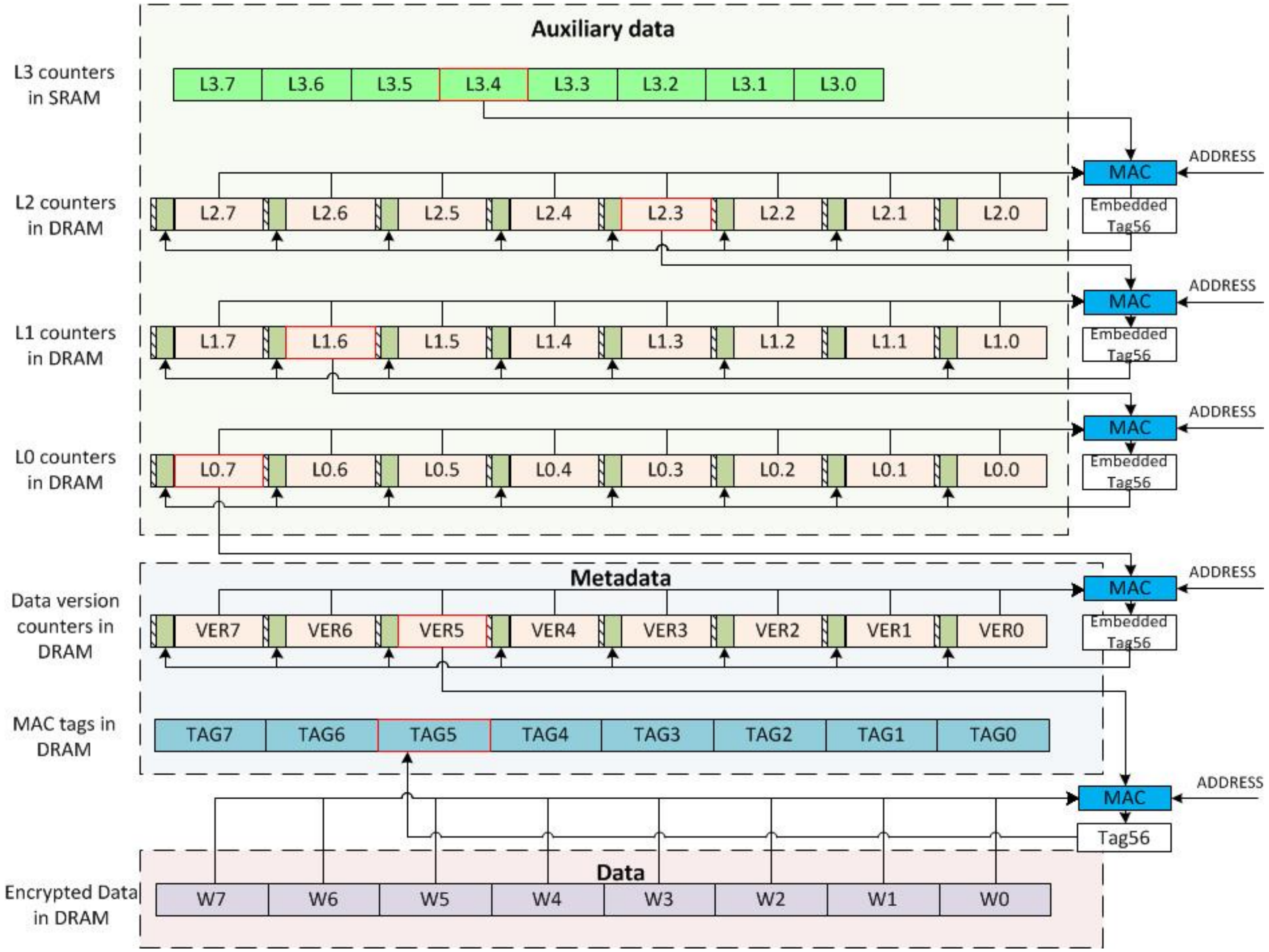
- DATA that is read back from DRAM to LLC is the same DATA that was most recently written from LLC to DRAM.

These properties address:

Data confidentiality, traffic analysis, random corruption, replay, cold boot attacks.

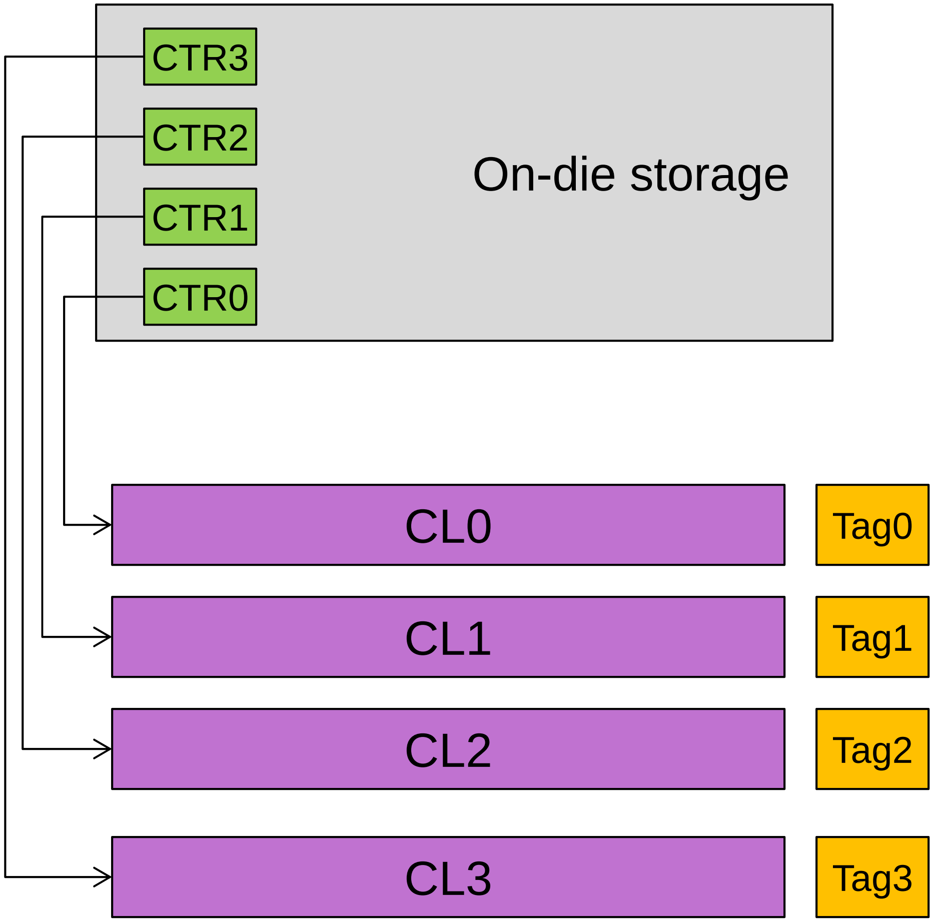
MEE does not hide the fact that data is written to the DRAM , when it is written, and to which physical address

The MEE data structure

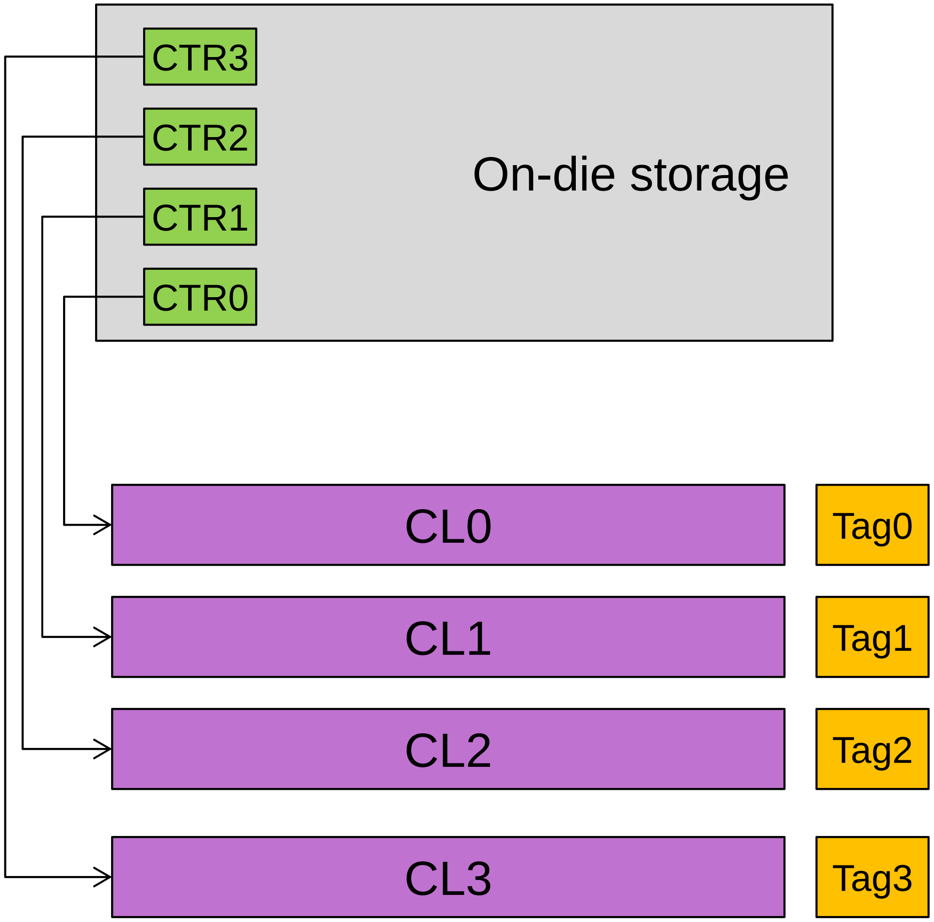


Better explained with a toy model

An abstract 1-level data structure



An abstract 1-level data structure

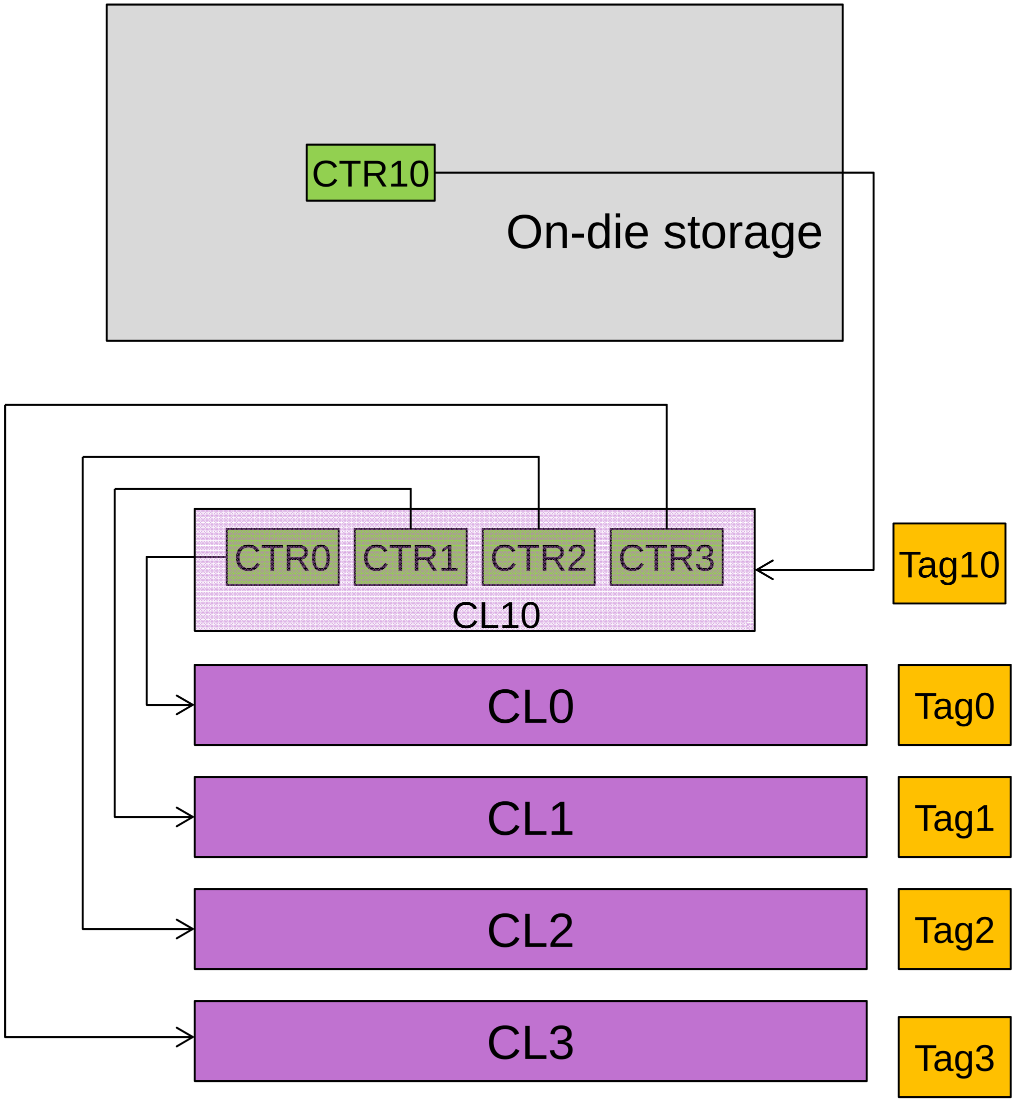


- In DRAM:
  - CL's (data)
    - Encrypted
  - MAC tag for each CL
- In internal storage (SRAM)
  - Counter for each CL
    - Incremented each write

- The MAC algorithm is "stateful" (i.e., uses the CTR value),
  - Preventing a "replay"

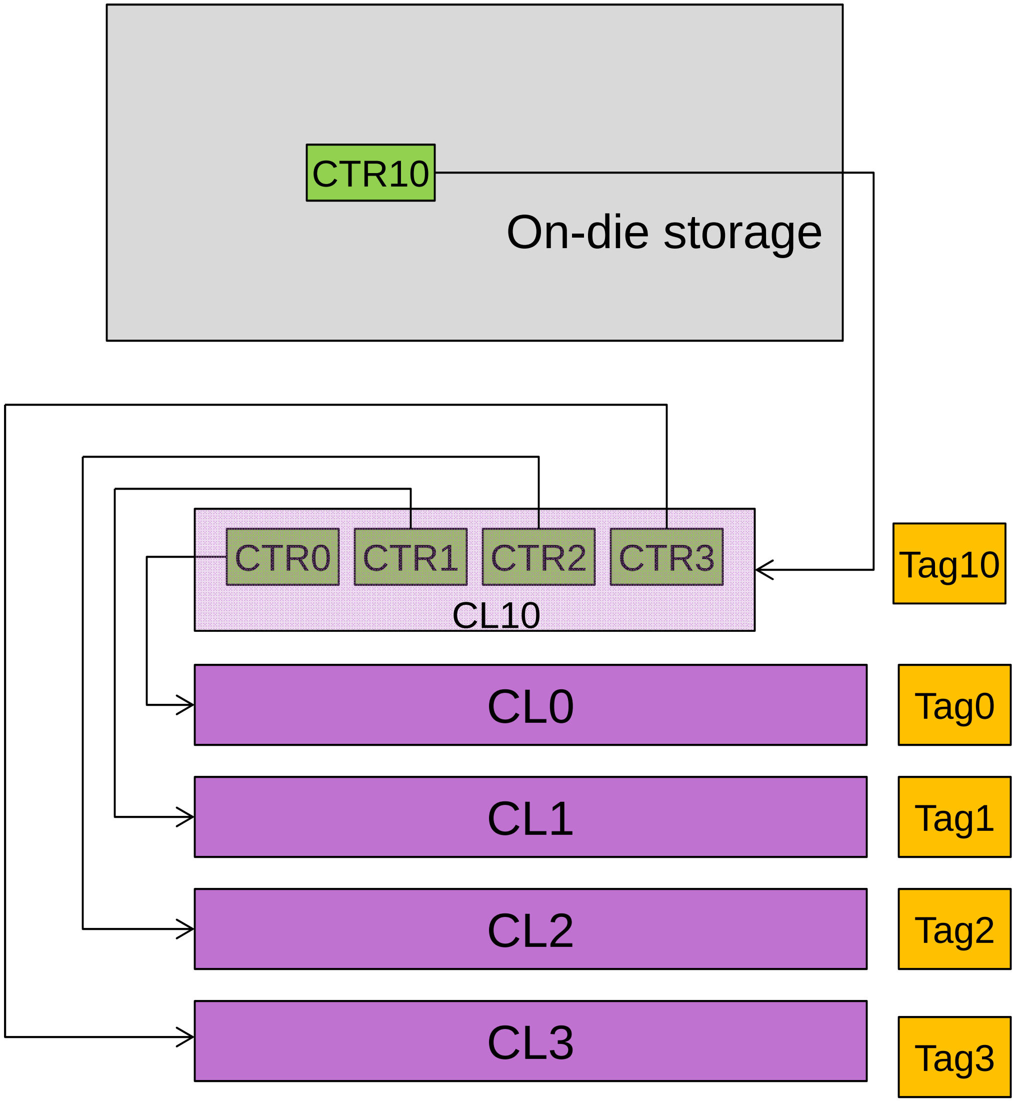
- Problem: large SRAM budget
  - SRAM is expensive

An abstract 2-level data structure



- In DRAM:
  - CL’s (data)
    - Encrypted
  - MAC tag for each CL
  - Counter per CL
    - Incremented per write
- In internal storage (SRAM)
  - 1 counter for “many CTR’s”
    - Incremented per write
- “stateful” MAC algorithm
  - i.e., uses the CTR
  - Addressing “replay”
- Tradeoff:
  - Less SRAM budget
  - Complexity & performance

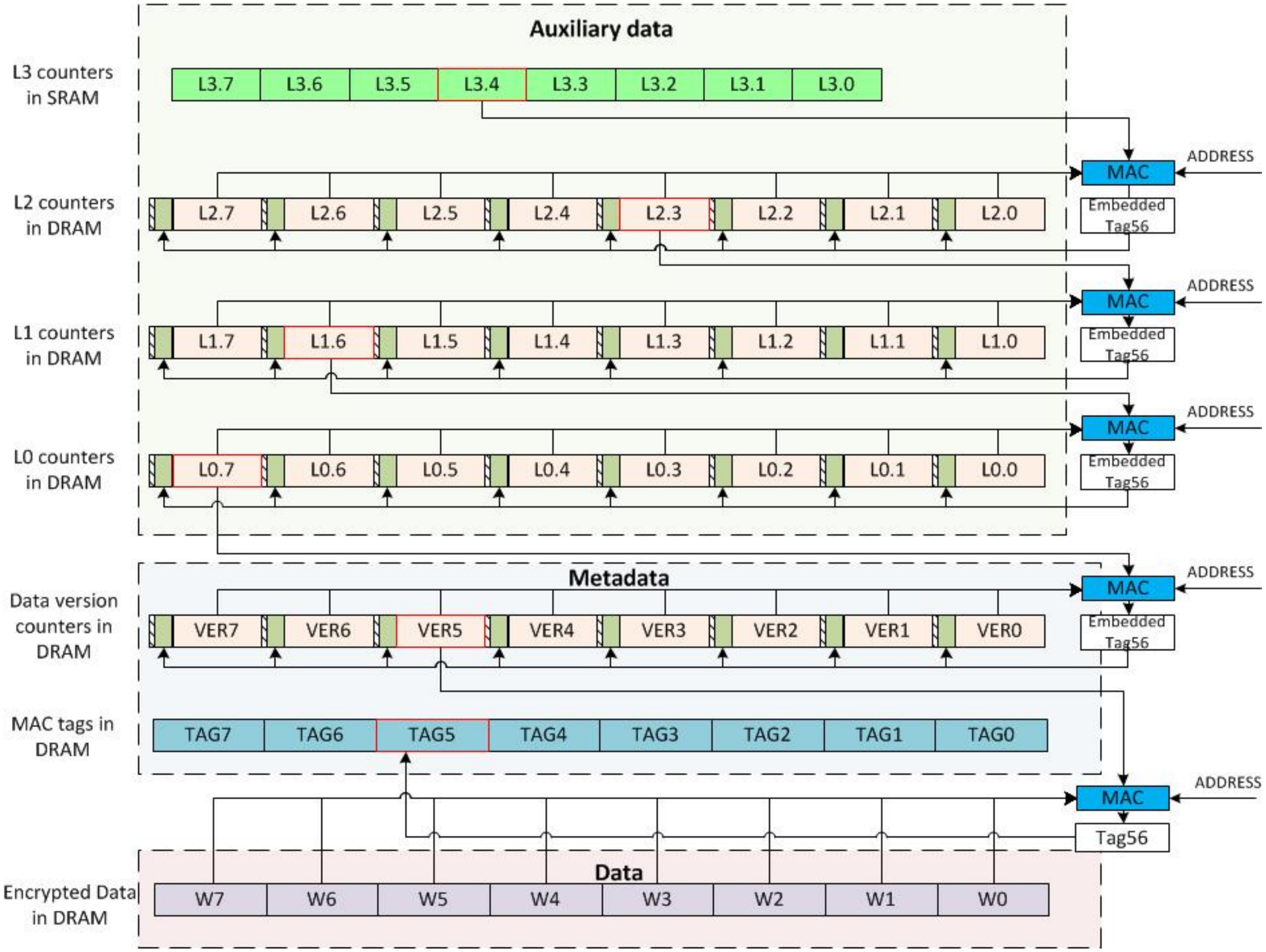
The 2-level data structure (2)



- (**obvious**) Read includes tag verify
- (**less obvious**) Write also includes MAC tag verification
  - Because forgery can touch data, tag, CTR on DRAM
- The "reduction argument"
  - Due to the unmodifiable top level CTR:
    - Successful forgery must solve the problem of modifying data and/or tag but not associated CTR
    - (…somewhere…)

But wait a sec –
how is it possible to start?

MEE multi level data structure



The MEE data structure is a multi level generalization of the 2-level structure

MEE self initialization

If write also includes MAC tag verification then

- How can the system survive the first write operation (after boot)?
- Without proper initialization, the **first write** access to the MEE region **will fail** and be trapped by the drop-and-lock

Problem: active initialization of the full region is too lengthy to be practical, and therefore MEE uses another method.

Solution: Self initialization

- Assign a special counter value "*1*" to indicate to the MEE that the first write has not yet occurred, and then it skips the MAC checks
- And initialize the "children" (in the data structure below) to the special value
  - Indicating that the children are un-initialized

Only the SRAM is actively initialized to *1*.

- During the write operations, the initialization propagates to the lower levels of the data structure

Data structure compression rate

expensive 😵

Internal (SRAM)

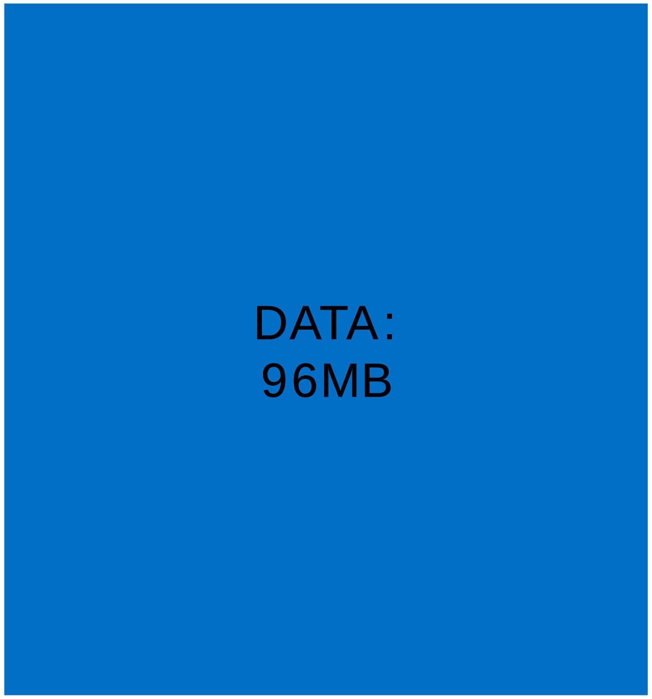
Memory (DRAM)

Data structure compression rate

expensive 😵

Internal (SRAM)

Memory (DRAM)



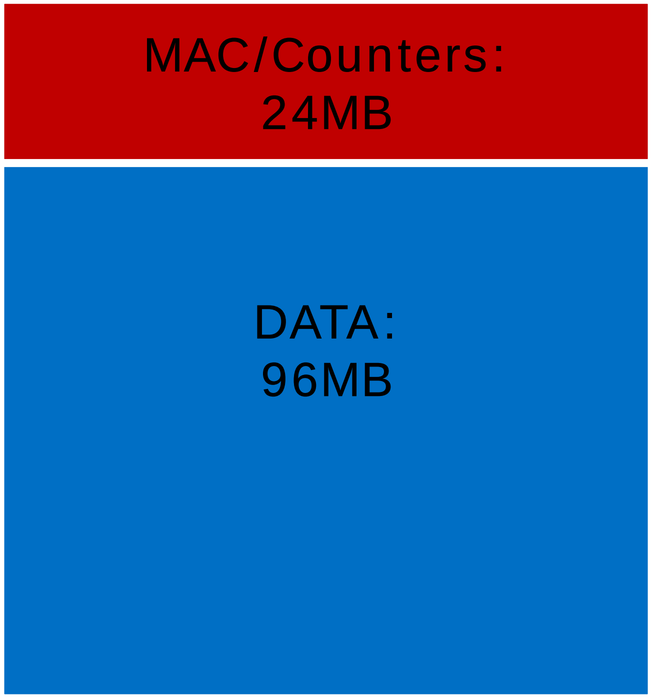
DATA:
96MB

Data structure compression rate

expensive

Internal (SRAM)

Memory (DRAM)



MAC/Counters: 24MB

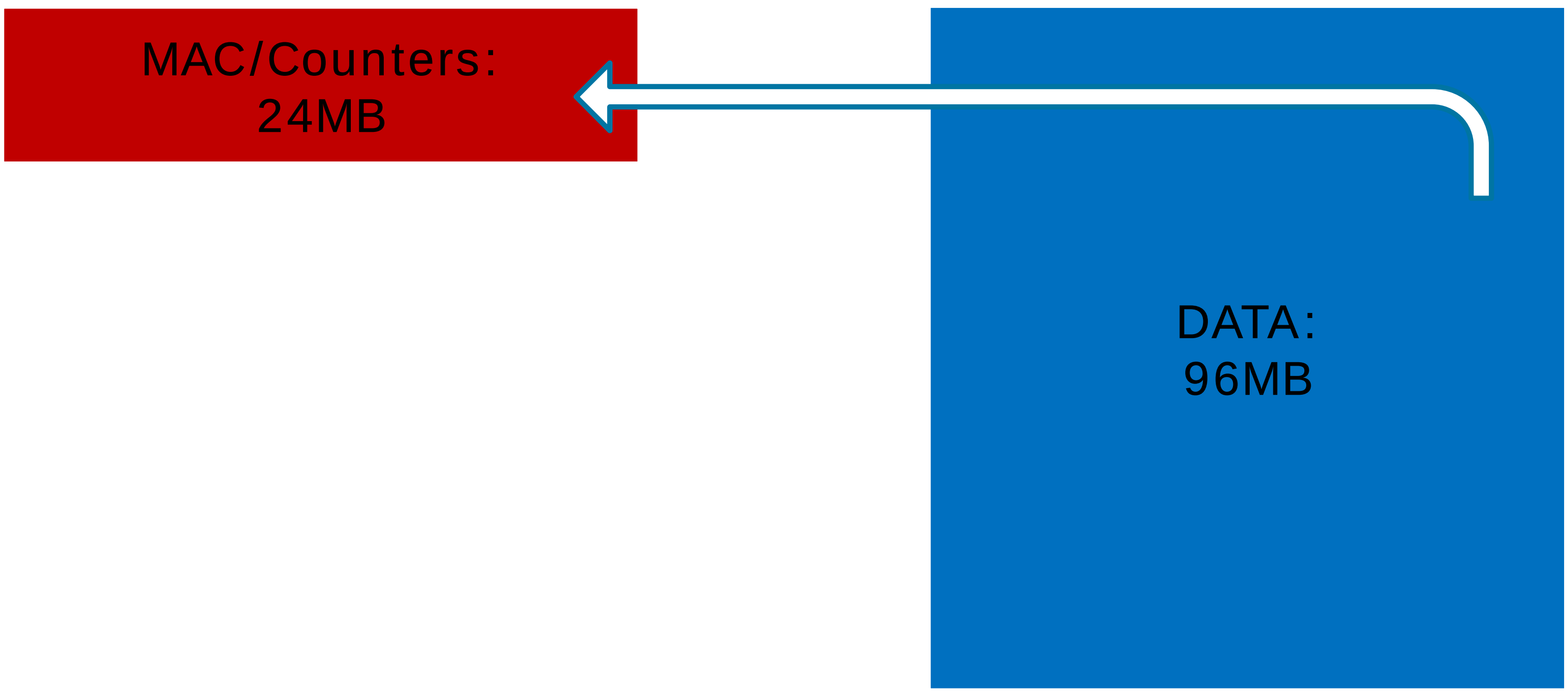
DATA: 96MB

Data structure compression rate

expensive 😵

Internal (SRAM)

Memory (DRAM)

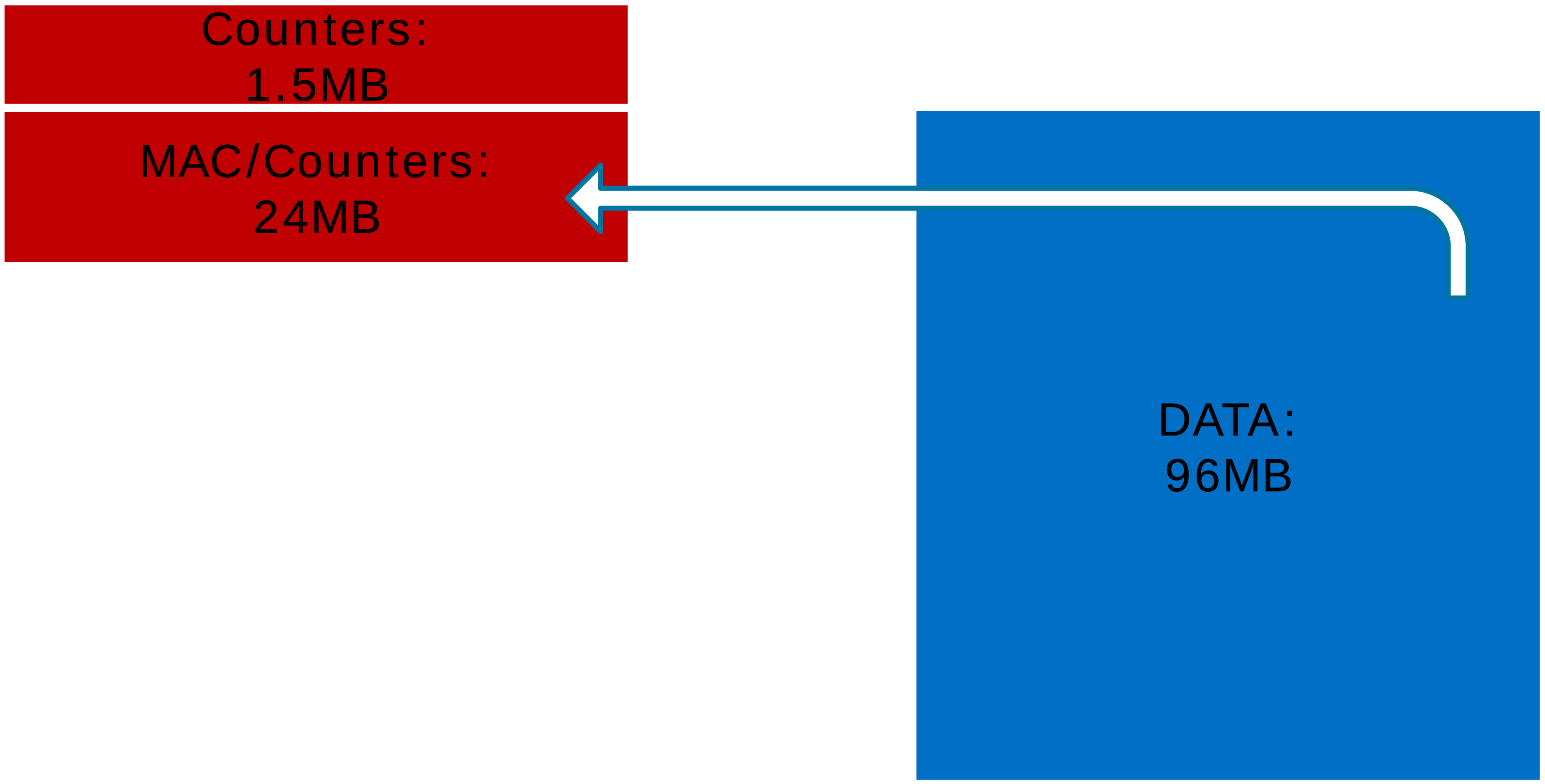


Data structure compression rate

expensive

Internal (SRAM)

Memory (DRAM)



Counters:
1.5MB

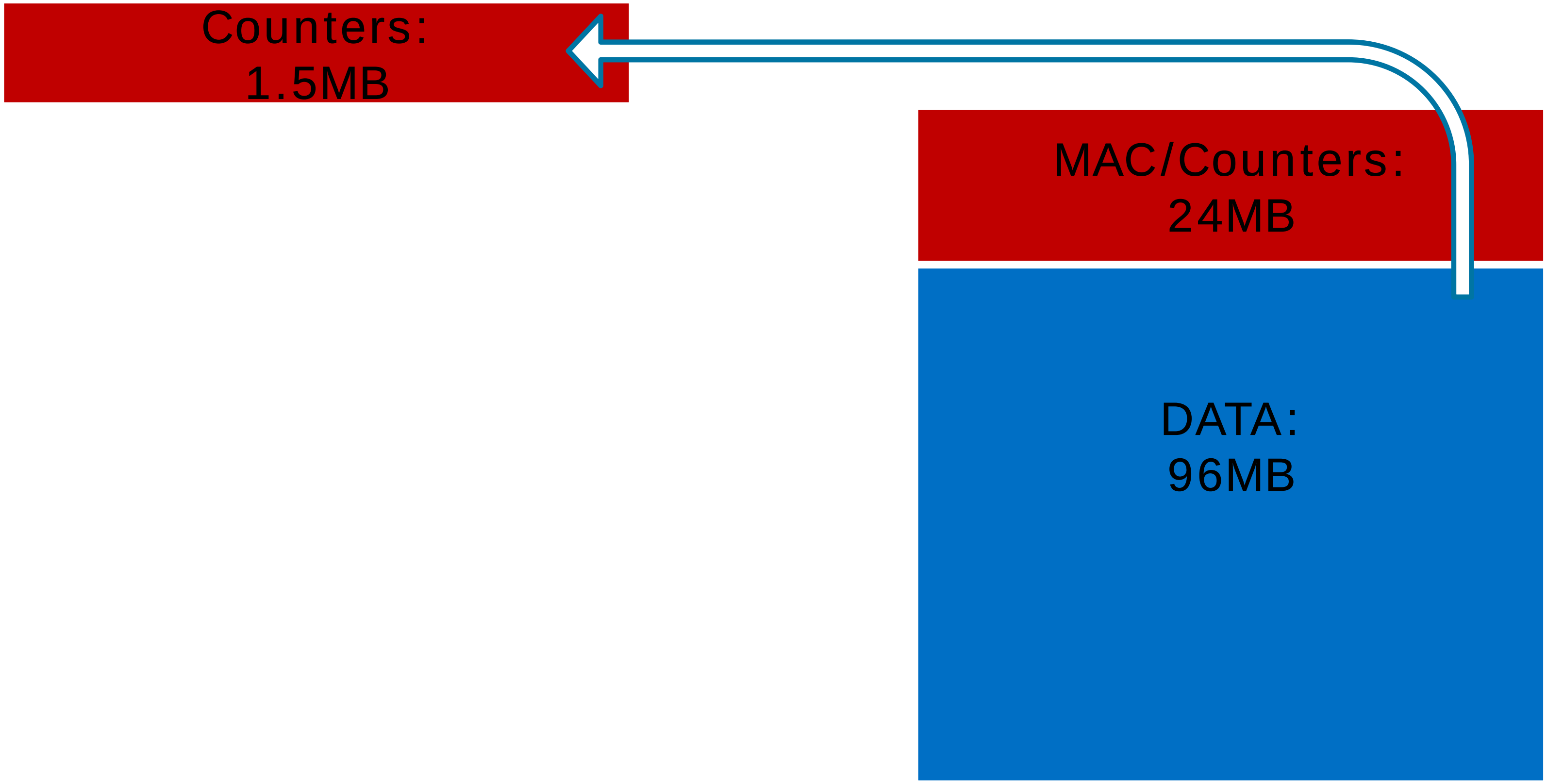
MAC/Counters:
24MB

DATA:
96MB

Data structure compression rate

expensive
😵

Internal (SRAM) Memory (DRAM)

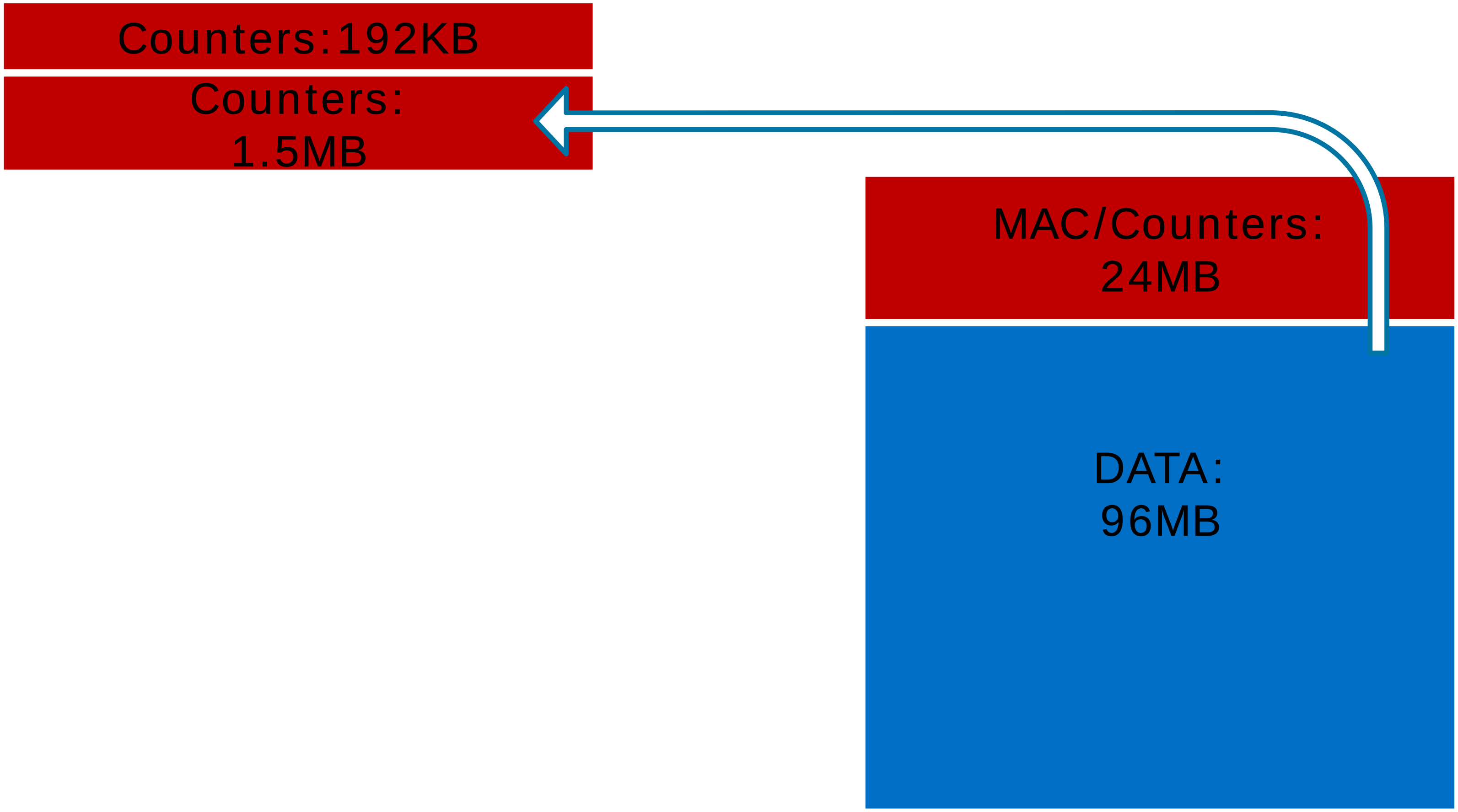


Data structure compression rate

expensive

Internal (SRAM)

Memory (DRAM)



Counters:192KB

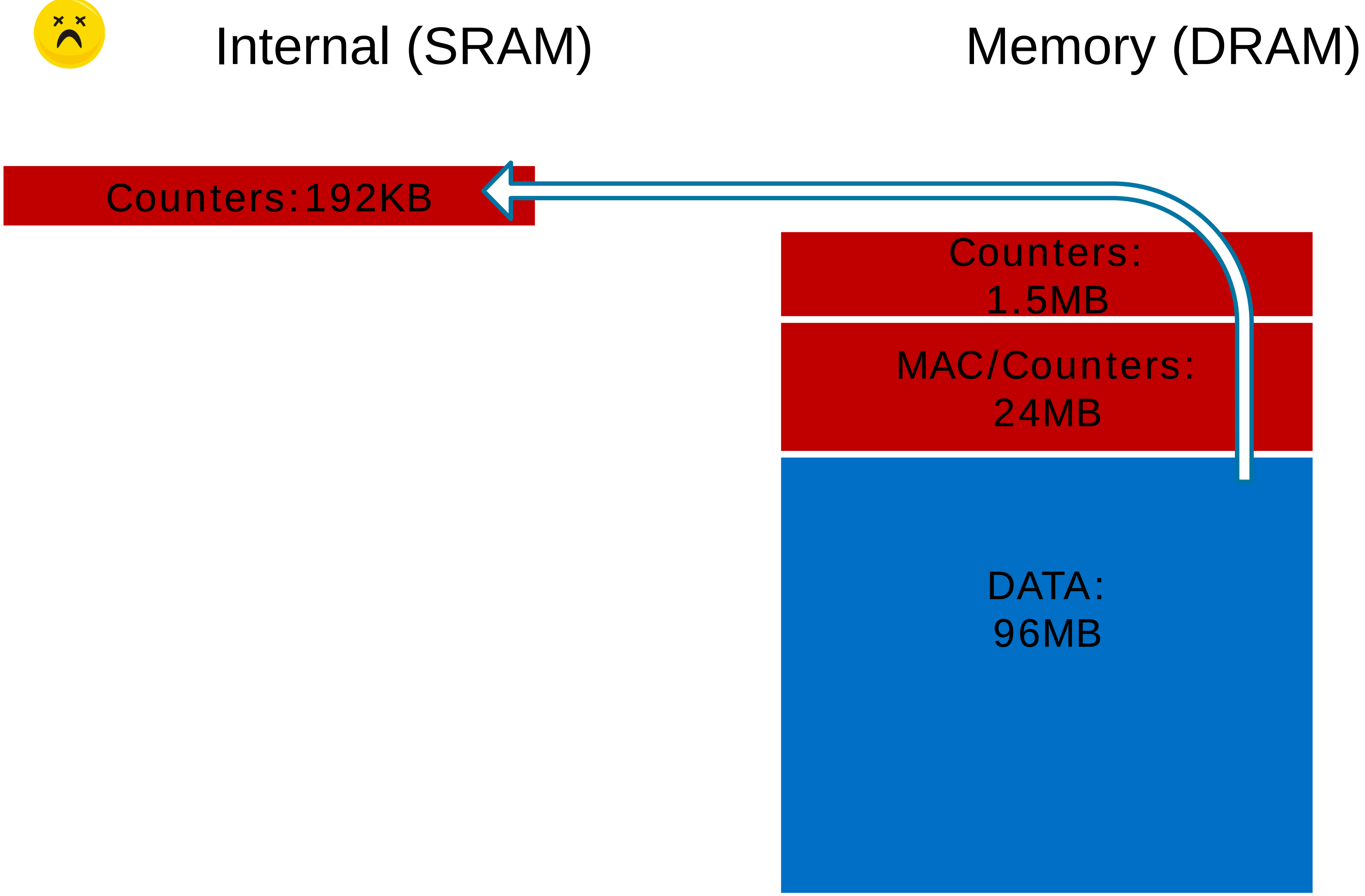
Counters: 1.5MB

MAC/Counters: 24MB

DATA: 96MB

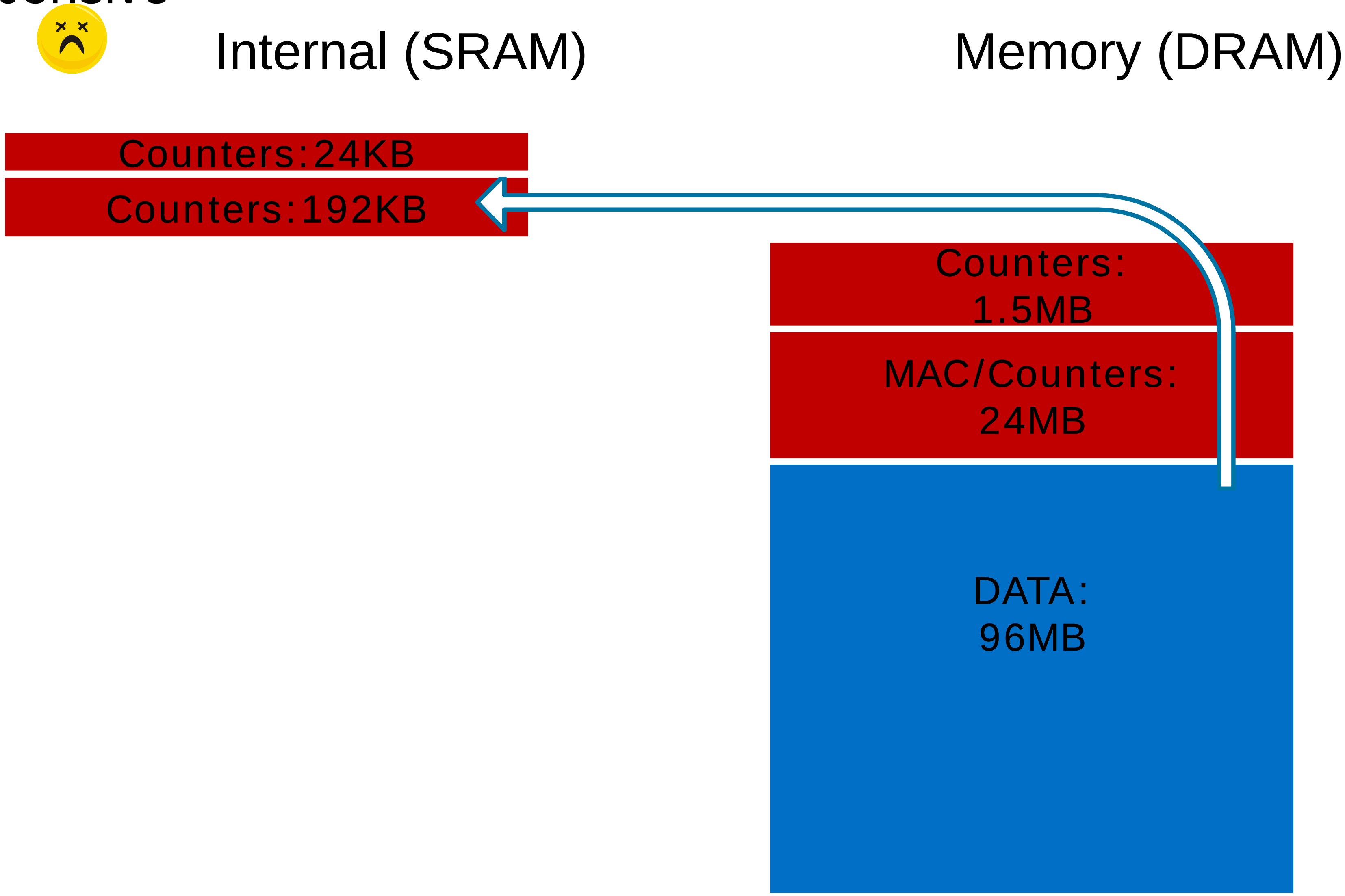
Data structure compression rate

expensive



Data structure compression rate

expensive



Internal (SRAM)

Memory (DRAM)

Counters:24KB

Counters:192KB

Counters: 1.5MB

MAC/Counters: 24MB

DATA: 96MB

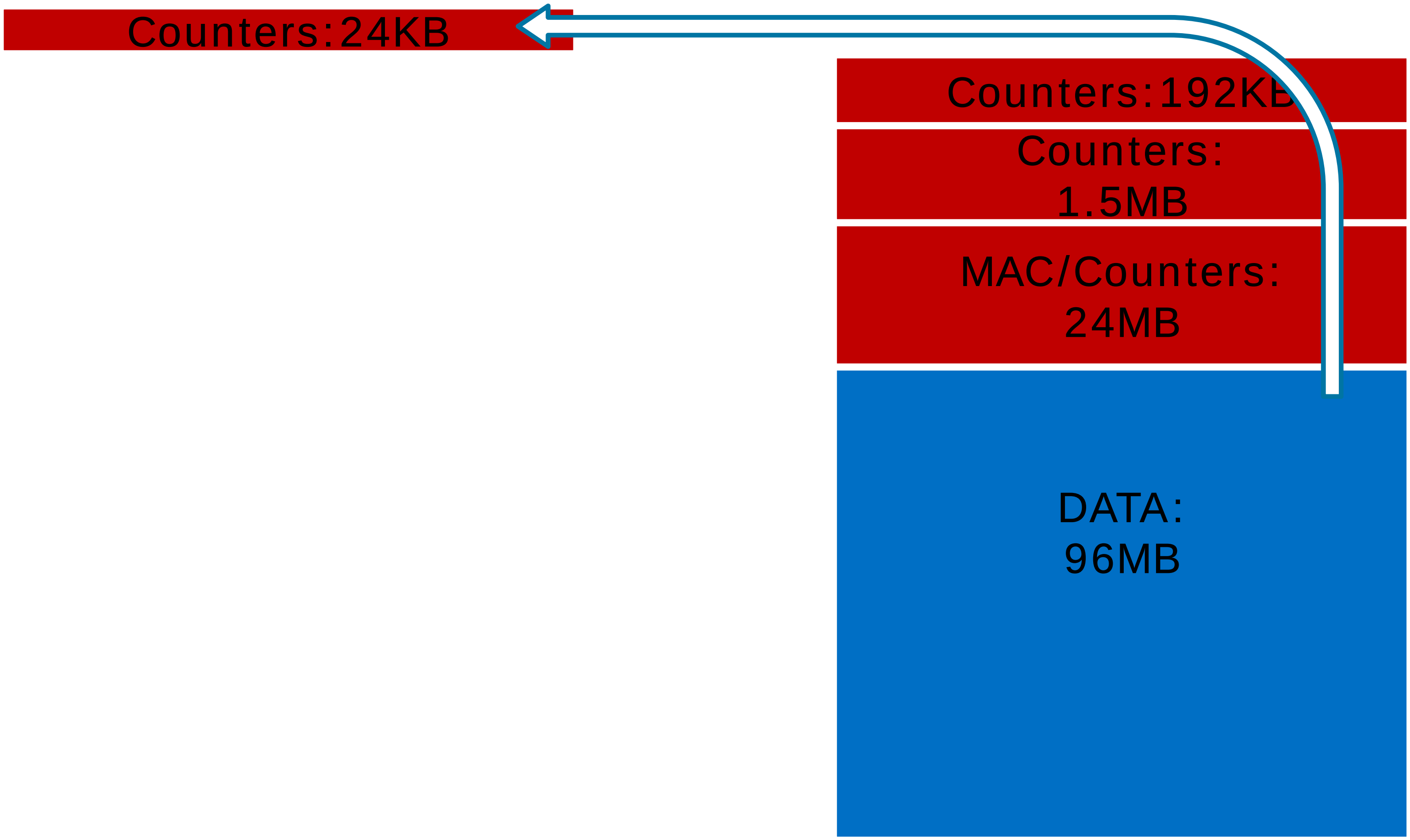
Data structure compression rate

expensive

Internal (SRAM)

Memory (DRAM)

Each additional level reduces SRAM requirement by a factor of 8



Counters:24KB

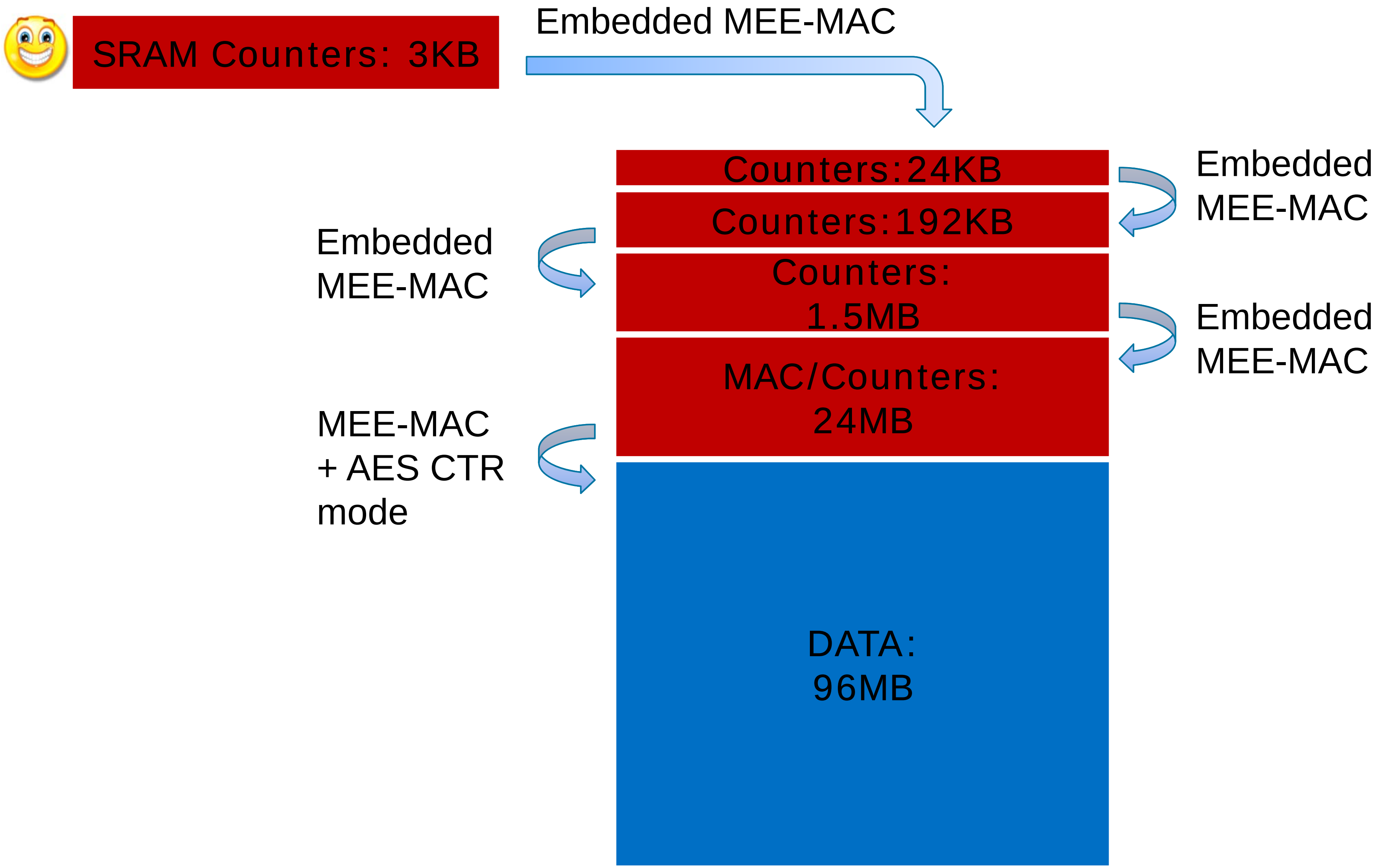
Counters:192KB

Counters: 1.5MB

MAC/Counters: 24MB

DATA: 96MB

Data structure compression rate



SRAM Counters: 3KB

Embedded MEE-MAC

Counters:24KB

Embedded
MEE-MAC

Counters:192KB

Embedded
MEE-MAC

Counters:
1.5MB

Embedded
MEE-MAC

MAC/Counters:
24MB

MEE-MAC
+ AES CTR
mode

DATA:
96MB

MEE keys and message space

Total key material: 768 bits

- Generated at boots, uniformly at random (by the DRNG unit)
- Placed in MEE registers

Split to 3 different keys

- Confidentiality key: 128-bit
- Integrity keys:
  - Masking key: 128-bit
  - Universal hash key: 512-bit

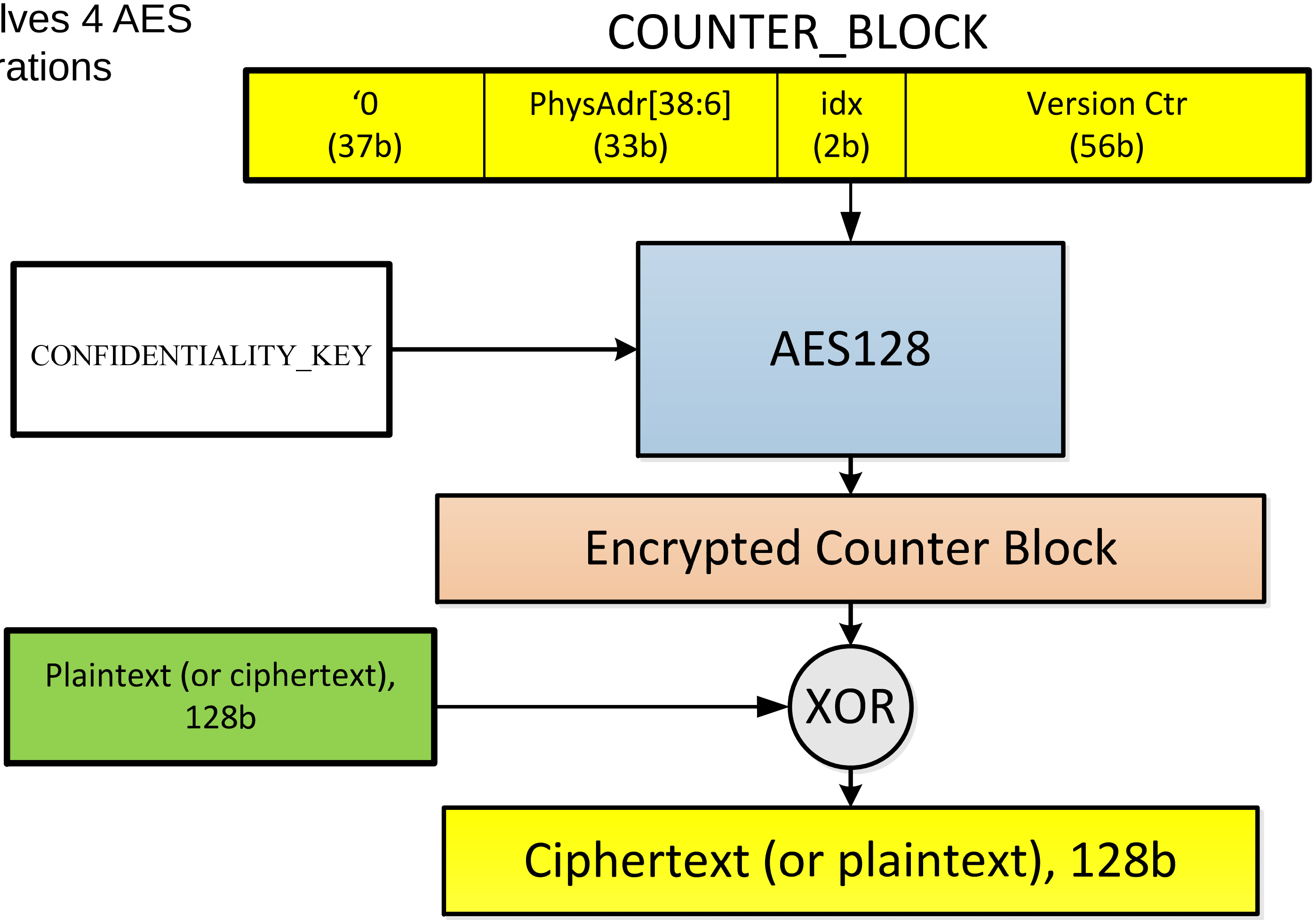
- Keys are never exposed outside the hardware
- Keys are destroyed at reset

- Encryption scheme operates on a 512-bit CL
- MAC scheme operates on a 512-bit CL

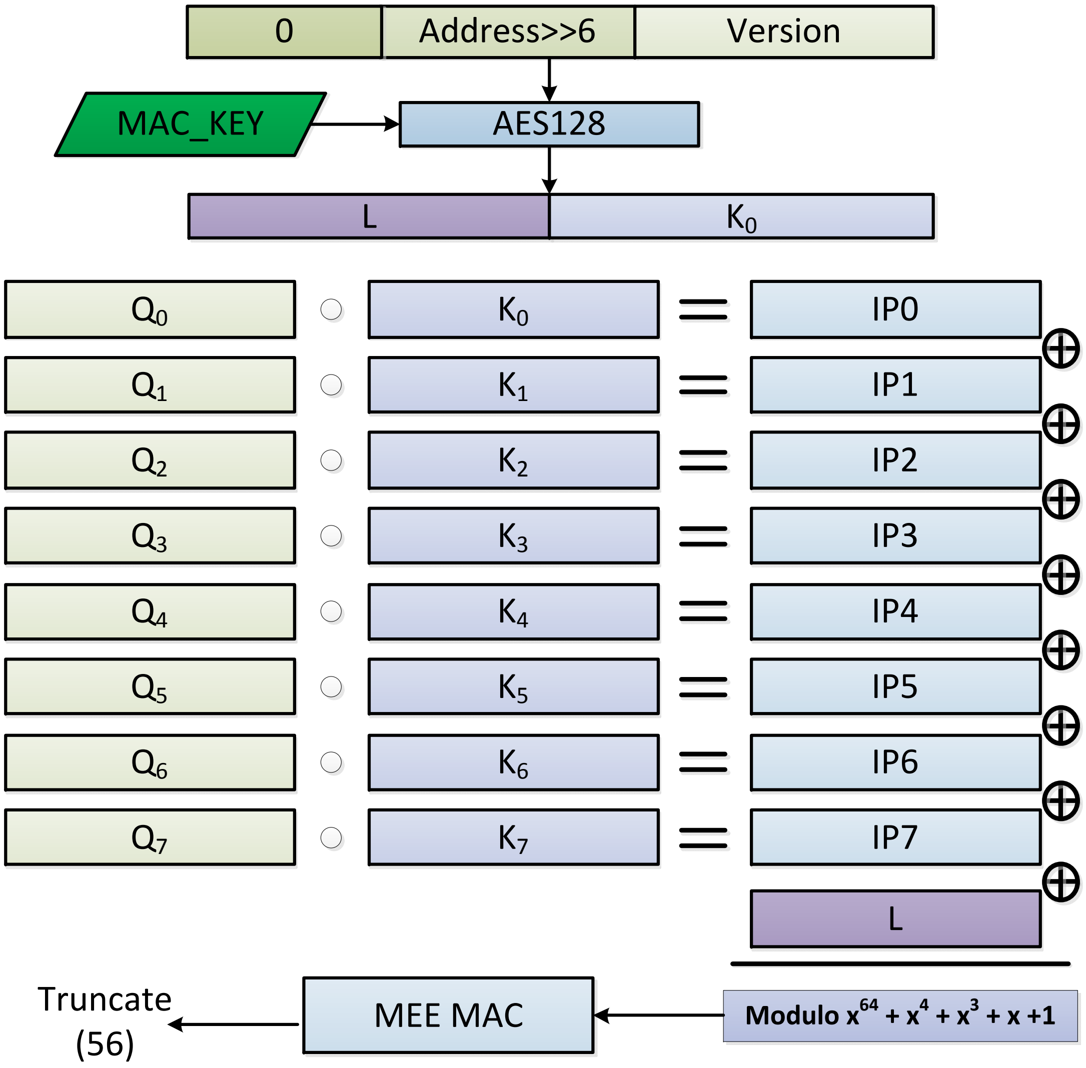
MEE Counter Mode

Address has 39 bits; idx: 2 bits representing location in the CL; Version: 56 bits

Encryption of 1 CL involves 4 AES operations



The MAC algorithm (2)



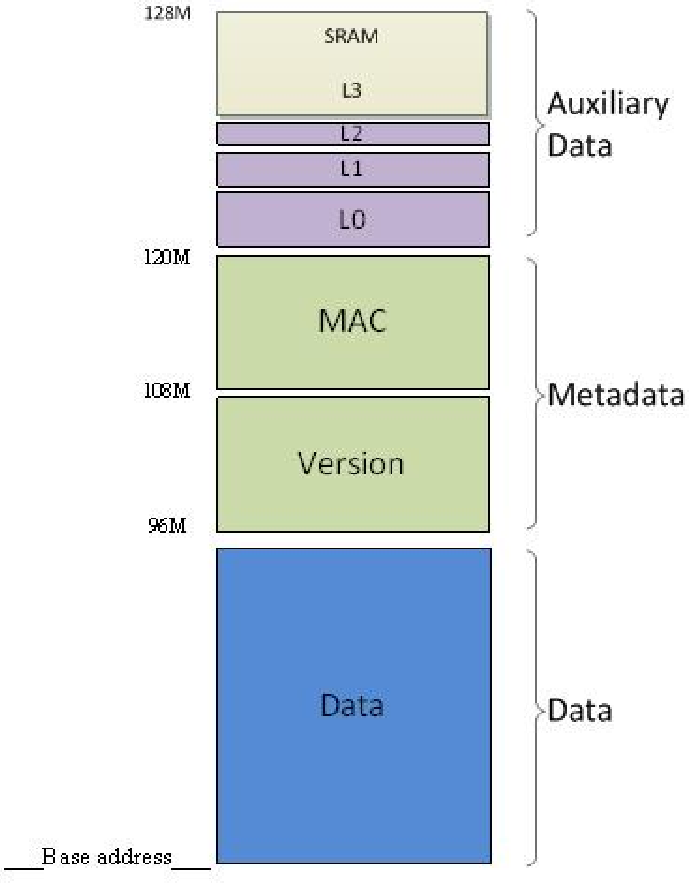
Multilinear universal hash

- (aka Inner “product hash”)
- Operations in GF (2^{64})

Masked by (truncated) AES

Truncated to 56 bits

The MEE data structure overall view



region is divided to sub regions:

ta (ciphertext CL’s)
rsion: temporal coordinate of CL
AC: MAC tags of data CL’s
, L1, L2, L3: counters with
bedded MAC tags

is stored on die (root of trust)

The MEE cache
Sweetening the performance degradation

Walking and processing the full read (write) flow on every cache miss can be very expensive

- 5 CL writes to the DRAM [DATA, MAC, Version, L0, L2, L2 (L3)]

But caching frequently used portions can significantly improve performance

- MEE cache hold counters and versions (not data)
- Counters retrieved from cache are not verified
  - Read/write flow stops at the cached node

- With a lucky MEE-cache hit at the lowest level: Read operation required only one decryption and one MAC operation

Some crypto theorems on security bounds

We can prove (information theoretic bounds) that (assuming AES is a good PRP)

- Collecting many samples (even 2^{56}) does not give a significant advantage in distinguishing MEE ciphertexts from random

- Collecting many MAC tags samples (even 2^{56}) does not improve the forgery success probability beyond $1/2^{56}$ in any meaningful amount

- At 2^{56} samples the game is over (drop-and-lock enforced)

How many samples can the adversary see?
Putting the crypto bounds to the test

- Can an adversary see 2^{56} ciphertexts?
- Can an adversary rollover 2^{56} counter?
- Can an adversary make ~2^{56} MAC tag guesses

- The answer depends on the real system's limitation
  - The throughput of the (fully pipelined) AES engine: 16B per cycle (at 2GHz)
  - CL ciphertext requires 4 AES operations
  - At minimum, 1 MAC tag is required → 1 more AES operation
  - At best 1/5 of the engine's throughput

How many samples can the adversary see?
Putting the crypto bounds to the test

- Assuming MEE cache hit on all of the MEE queries
- Assuming an adversary can make 1000 “forge-boot” attempts per sec
- (dedicated attack machine)

- Rollover (serial) would take ~5 year
- Forgery (parallelizable) would take ~2M years
  - (or, 2 years over 1M machines doing forge-boot constantly)

Conclusion

- Real-word memory encryption is a huge engineering challenge
  - MEE is essential to Intel® SGX technology
- MEE is based on
  - A careful combination of cryptographic primitives
  - And efficient data structure
  - Designed to achieve the cryptographic properties, while satisfying very strict engineering constraints
- MEE provides data confidentiality and integrity (in the broad sense)
- Robustness:
  - Rollover (serial attack) would take ~5 year
  - Forgery (parallelizable) would take ~2M years
    - (or, 2 years over 1M machines doing forge-boot constantly)

Implementation of SGX on Intel® Core™ Processors

The Challenges

How do you even start designing SGX on a CPU?

- Security requirements are so strong
  - And yet we need to be able to debug the part
- Complexity is very high
  - Number of new instructions (18)
  - Number of the pages in the PRM (186)
  - Complexity of each instruction (5 pages on average)
    - Lead into complex micro-architectural interactions.
  - Interaction with so many components (provisioning infrastructure, BIOS, System SW, application SW)

Close Collaboration

Close collaboration between micro-architecture team and ISA team

- Co-development of the ISA
  - ISA went through changes and reduction in order to meet project design scope.
- Design-friendly options were more favorable
  - TCB recovery could be much complex
- Implementation team were up to date on the ISA
- Close relationship lead to faster turnaround times

Security

We started the Security Development Lifecycle (SDL) very early

- Identified the security assets (keys and secrets) from the beginning
- Identified the adversaries very early so that we knew who to protect against
- Defined the micro architectural security objectives and design the proper debug utilities

With such a complex architecture we knew that **recoverability** must be tightly integrated into the design from the beginning

- Microcode updates are an integral component of recovery

Validation

We expanded our validation to beyond the traditional functional testing

- Security testing
  - Special teams focused on validating the design against the security objectives
  - Reserved, unused, not supposed to happen in real life scenarios – those were the most interesting
- Formal verification
  - Deployed various formal modeling techniques to prove security objectives of both the Spec (ISA) and the design
  - Big focus on proving resistance to multi-threaded race conditions

Shift Left

Given the huge amount of dependency on other components besides the CPU, we created simulation environments and built special HW to **shift left** (i.e. pull-in) development and get it ready for first silicon's arrival

- SDK
- Application SW
- System SW
- Mockup infrastructure environment for the provisioning components
- Testing infrastructure and content

Summary

SGX is a very challenging architecture to implement

- It must be secure and meet the security objectives
- It must be implementable
- It must be debug-able

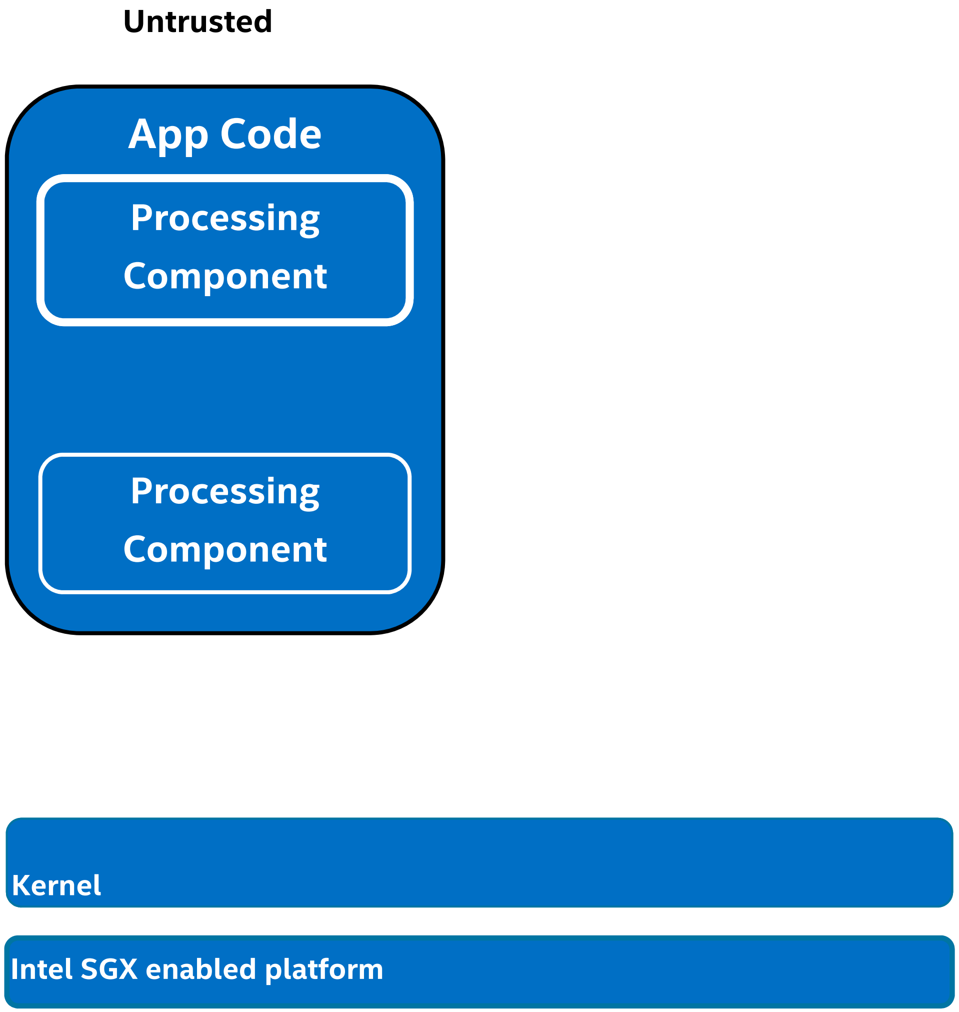
We're using both process and innovative techniques to get the job done on time with high quality

Software: Run Time Environment, EPC Management, and SDK for Linux

Introduction

- From previous sessions we learned that Intel SGX:
  - Provides a Trusted Execution Environment within application address space.
  - Allows developers to partition their application and move sensitive code and data into the TEE (or Enclave).
  - Protects user code and data even in the event of a compromised OS or VMM.
- This session will cover:
  - The SGX Application Development Model
  - An overview of potential components of the SGX Software Stack on Linux.

Intel® SGX Software Development Model



Untrusted

App Code

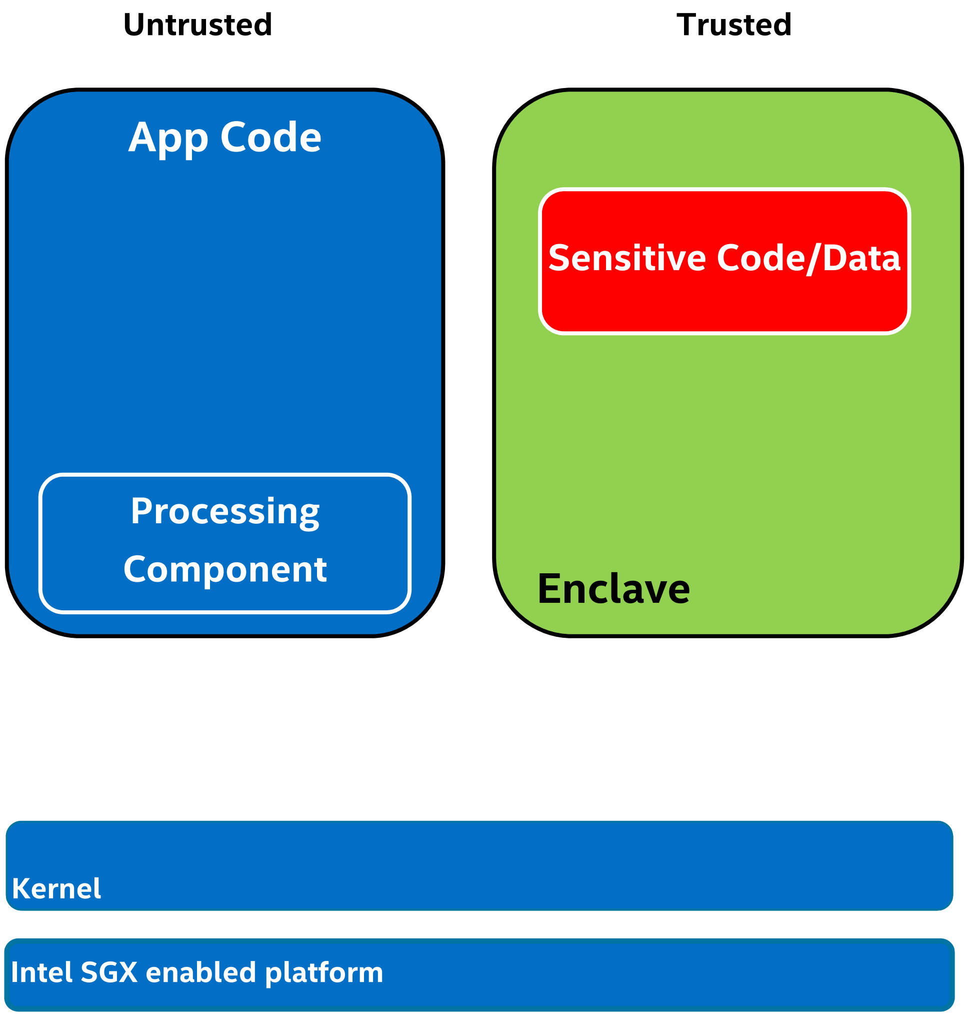
Processing Component

Processing Component

Kernel

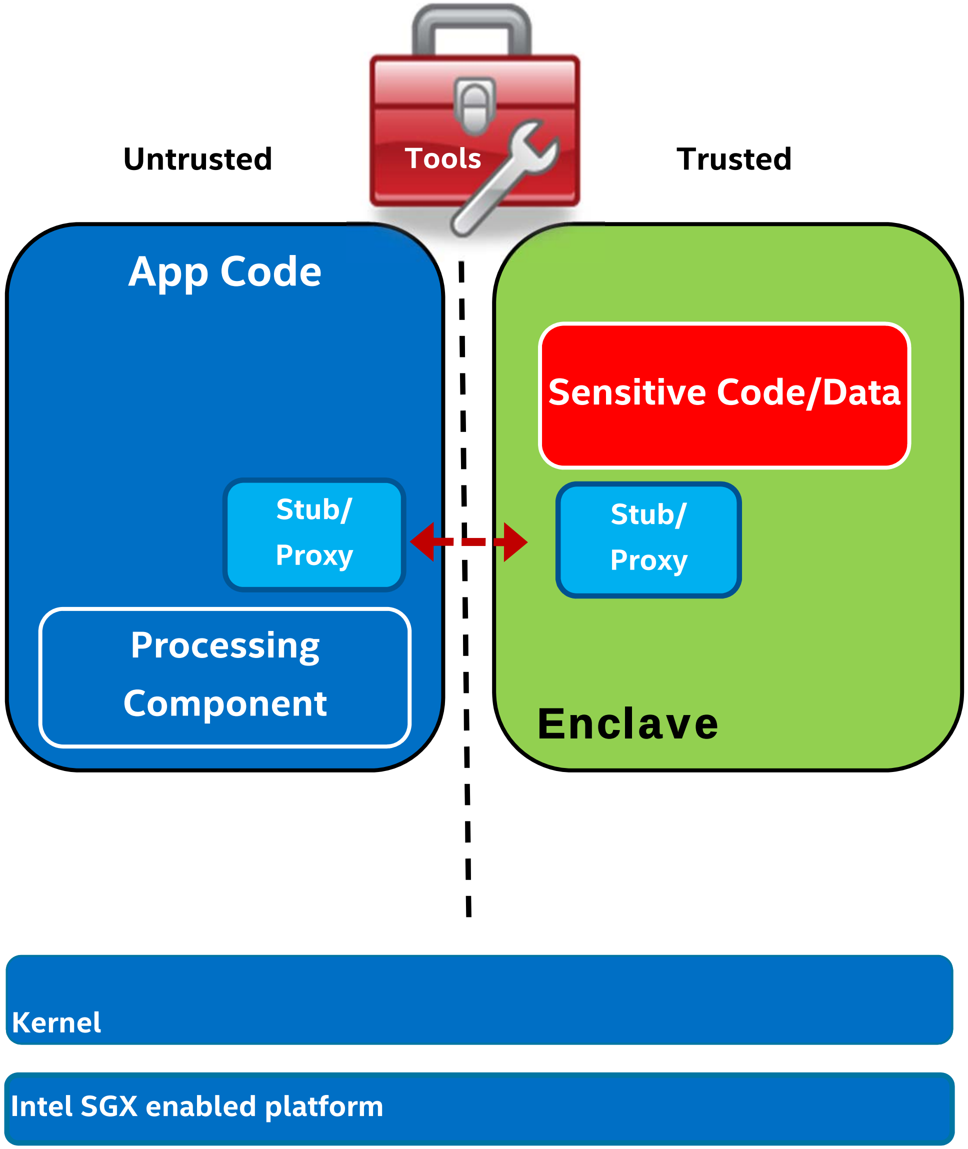
Intel SGX enabled platform

Intel® SGX Software Development Model



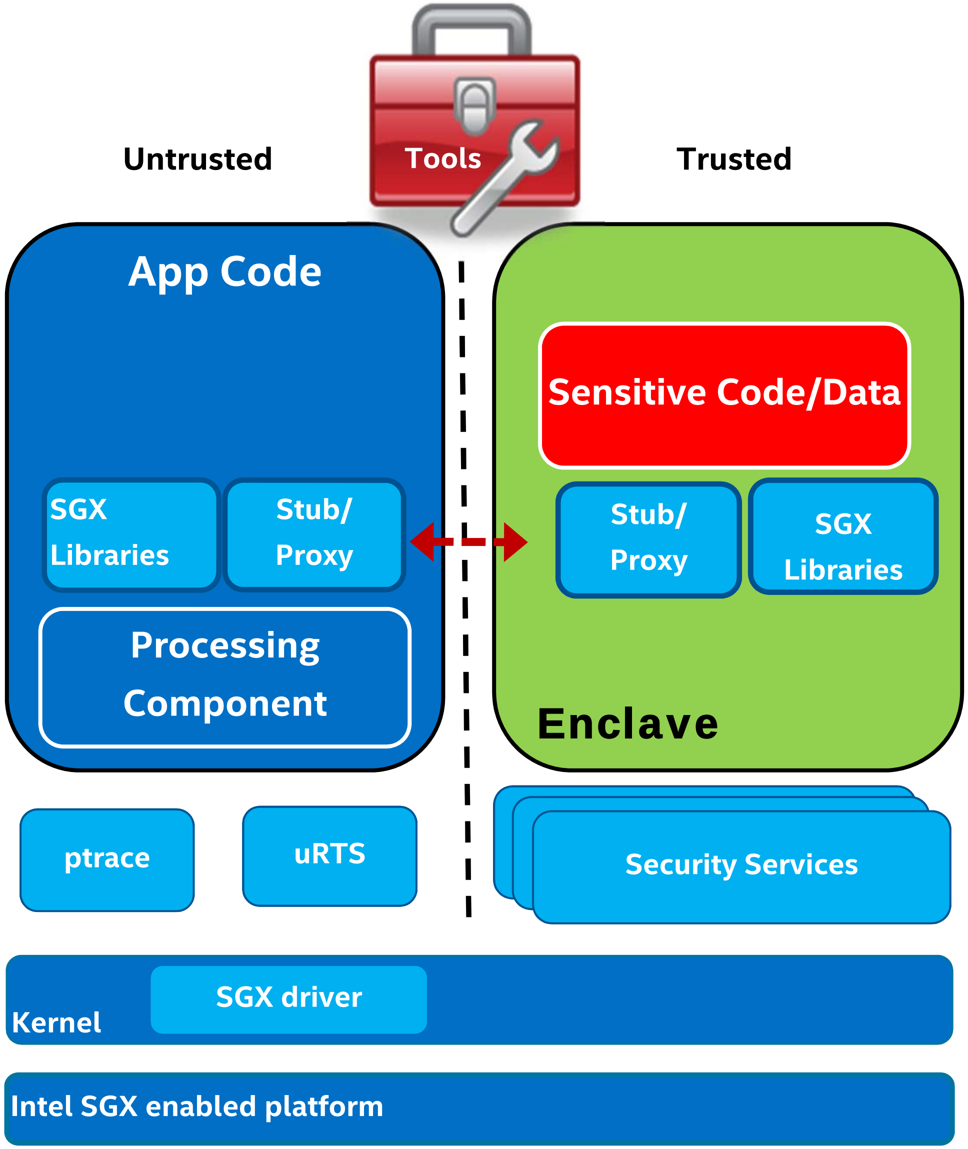
- Sensitive code and data is partitioned into an “enclave” module which is a shared object (.so)

Intel® SGX Software Development Model



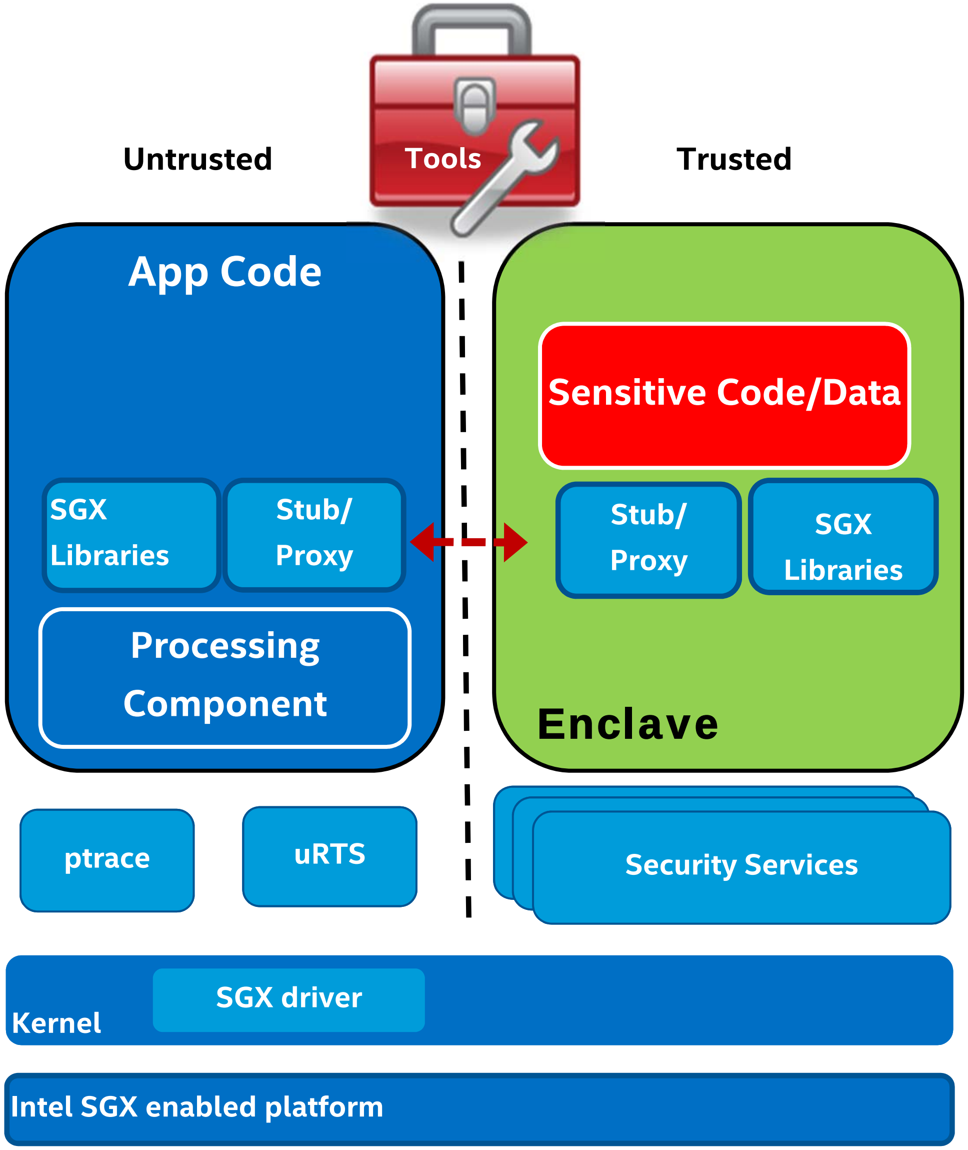
- Sensitive code and data is partitioned into an “enclave” module which is a shared object (.so)
- Define the enclave interface and use tools to generate stubs/proxies

Intel® SGX Software Development Model



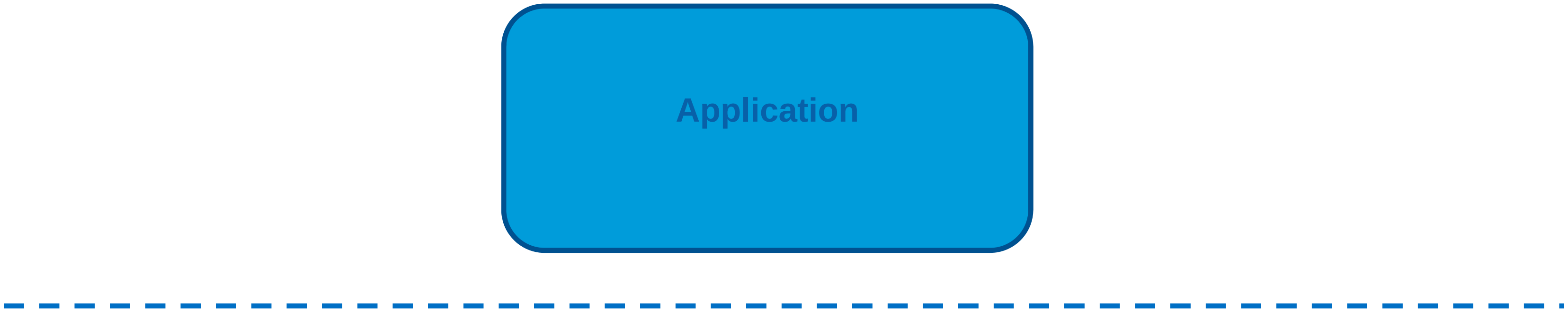
- Sensitive code and data is partitioned into an “enclave” module which is a shared object (.so)
- Define the enclave interface and use tools to generate stubs/proxies
- SGX Libraries provide APIs (C/C++) to encapsulate heavy-lifting implementation

Intel® SGX Software Development Model



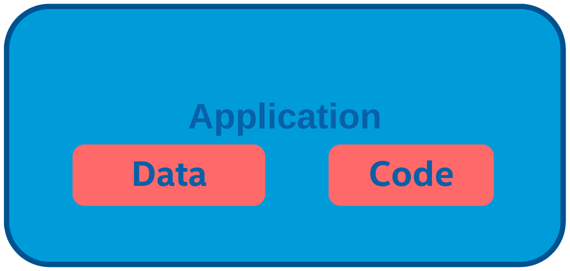
- Sensitive code and data is partitioned into an “enclave” module which is a shared object (.so)
- Define the enclave interface and use tools to generate stubs/proxies
- SGX Libraries provide APIs (C/C++) to encapsulate heavy-lifting implementation
- Use a familiar toolchain to build and debug

Hardening an Application with SGX



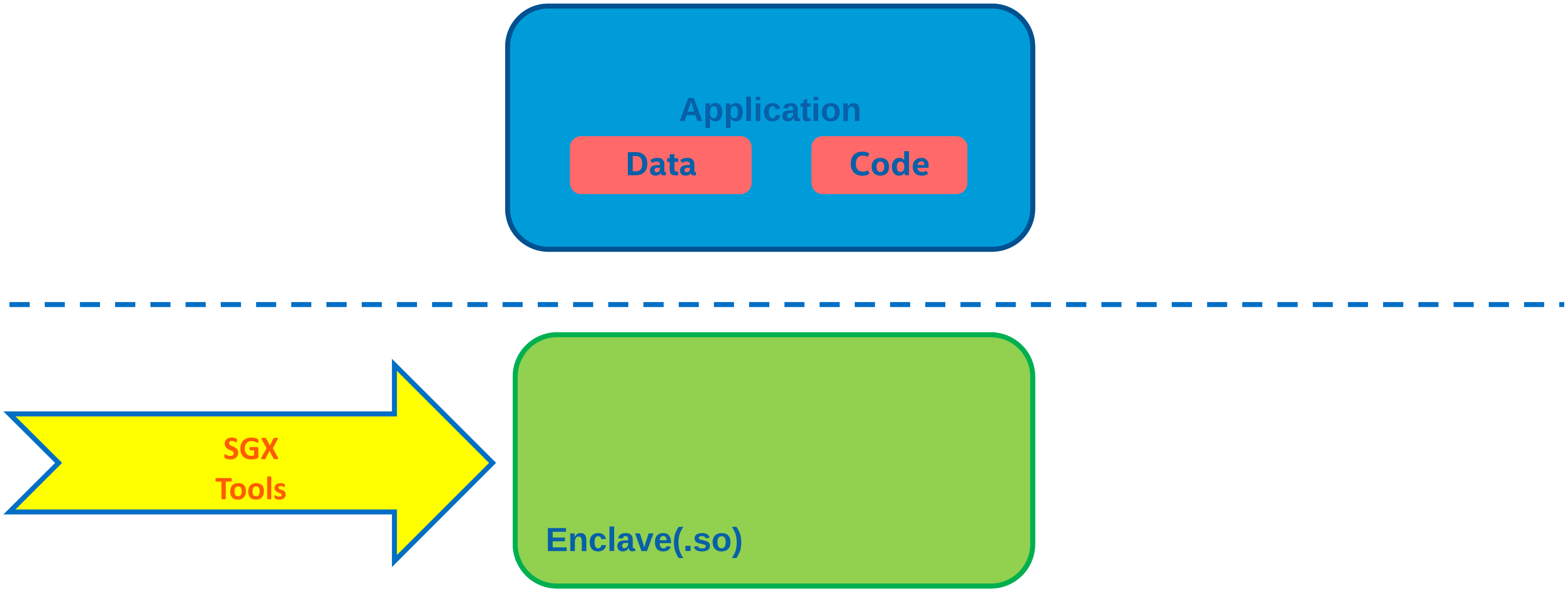
Application

Hardening an Application with SGX



1. Identify Sensitive Data Objects and Code

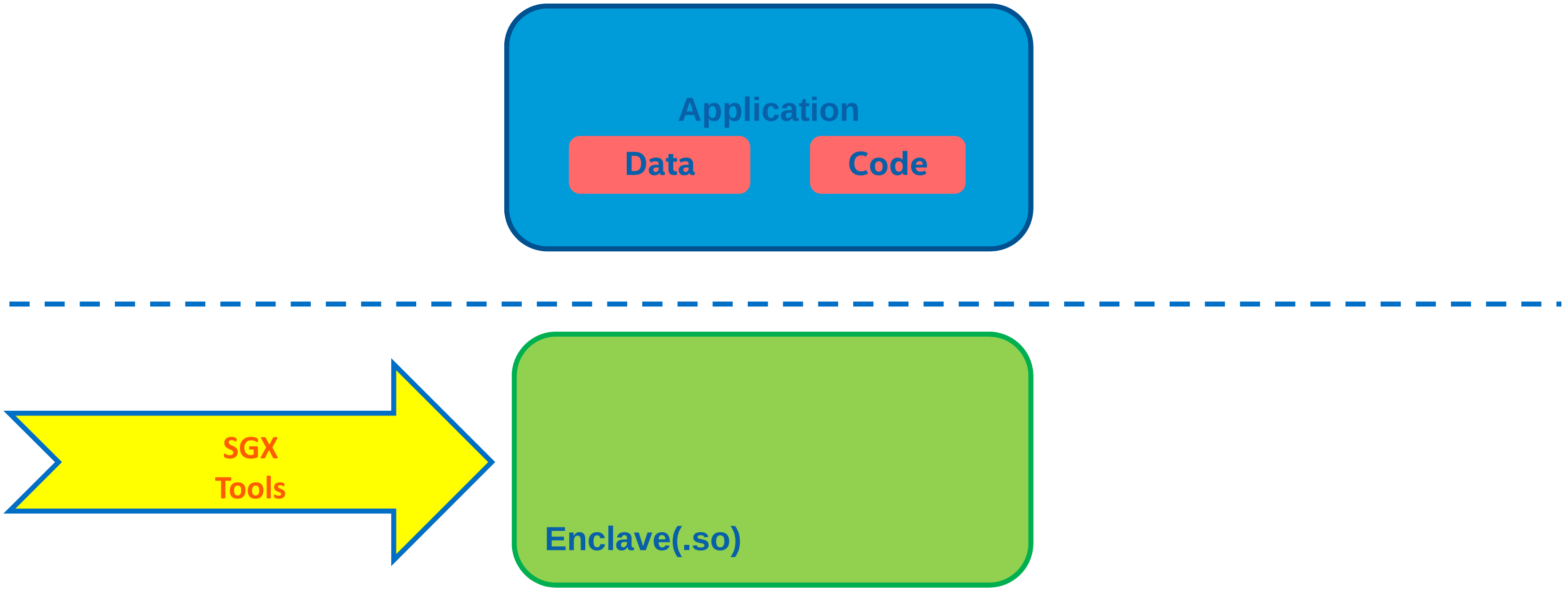
Hardening an Application with SGX



1. Identify Sensitive Data Objects and Code

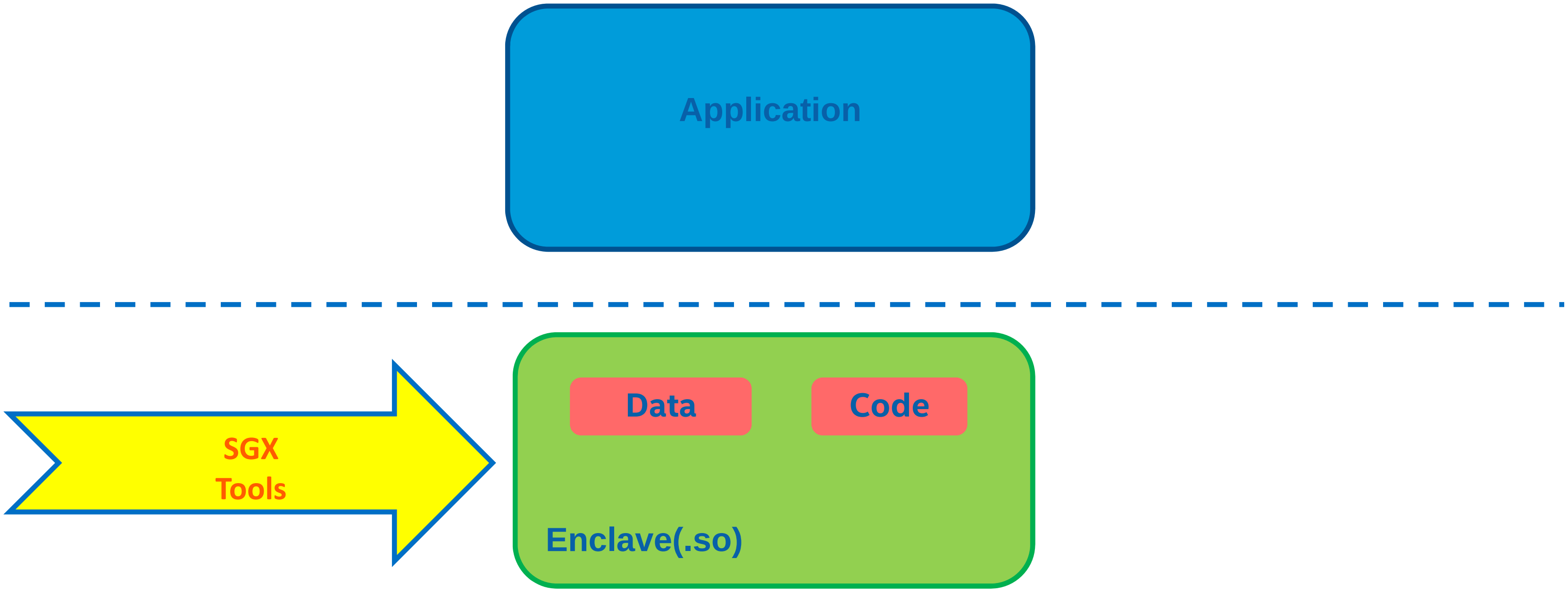
2. Use SGX Tools to create an Enclave Module (Shared Object)

Hardening an Application with SGX



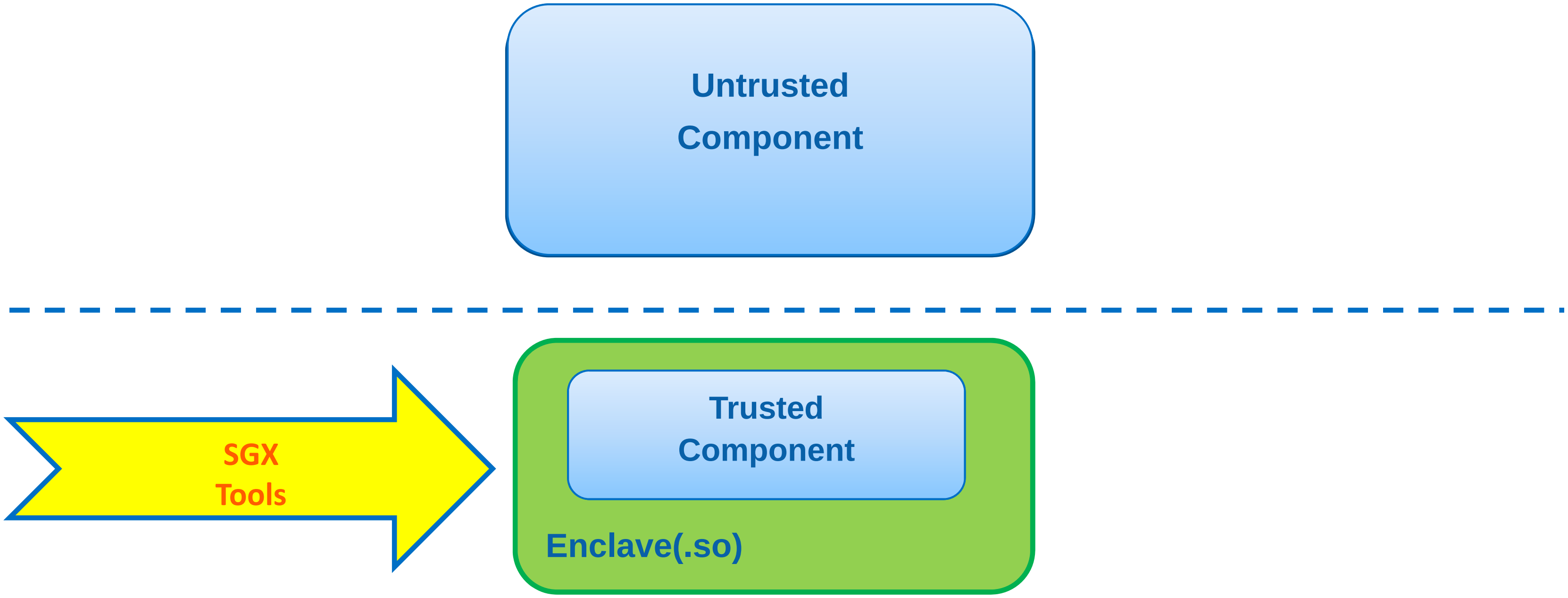
1. Identify Sensitive Data Objects and Code
2. Use SGX Tools to create an Enclave Module (Shared Object)
3. Move Sensitive Data Objects and Code to the Enclave

Hardening an Application with SGX



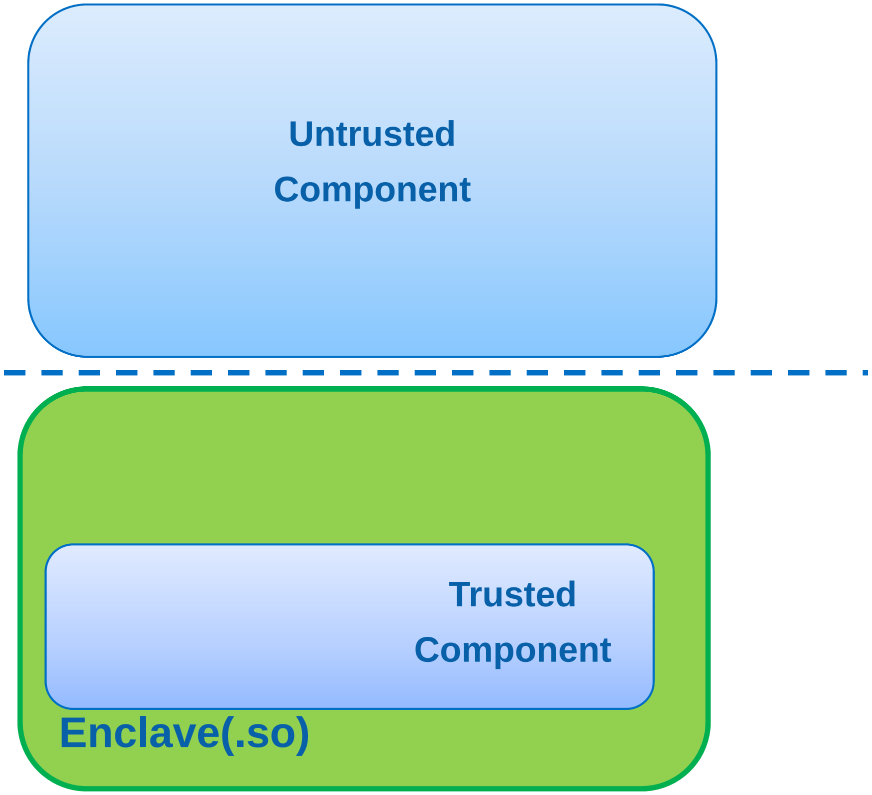
1. Identify Sensitive Data Objects and Code
2. Use SGX Tools to create an Enclave Module (Shared Object)
3. Move Sensitive Data Objects and Code to the Enclave
   - Change only affects Application Modules which host sensitive data and code.

Hardening an Application with SGX

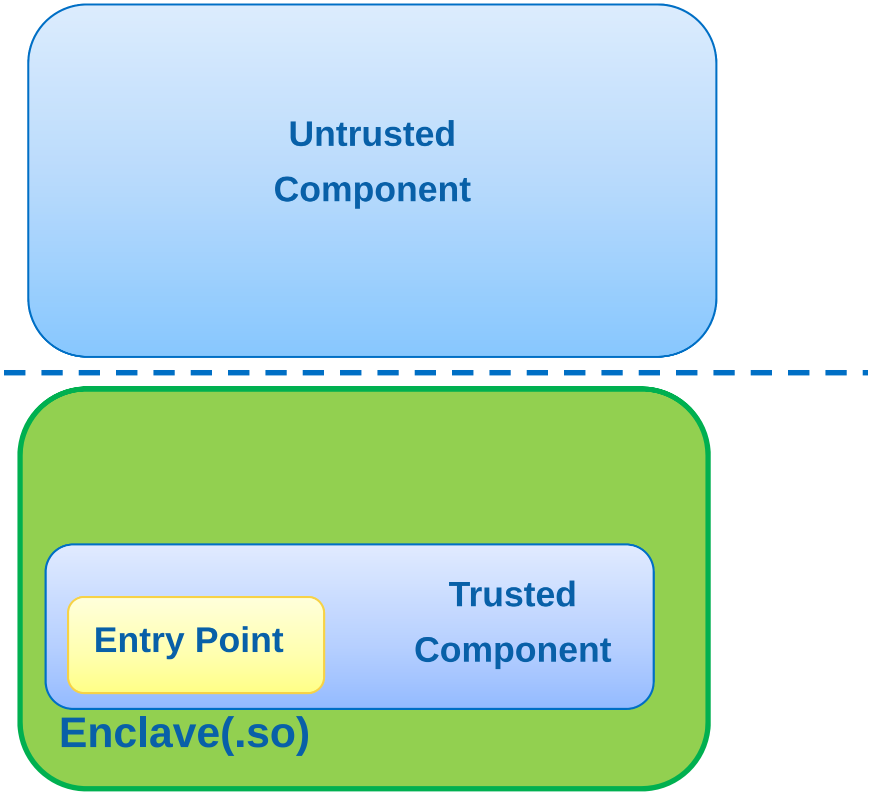


1. Identify Sensitive Data Objects and Code
2. Use SGX Tools to create an Enclave Module (Shared Object)
3. Move Sensitive Data Objects and Code to the Enclave
   - Change only affects Application Modules which host sensitive data and code.
   - Application now consists of an Untrusted Component and a Trusted Component.

Hardening an Application with SGX

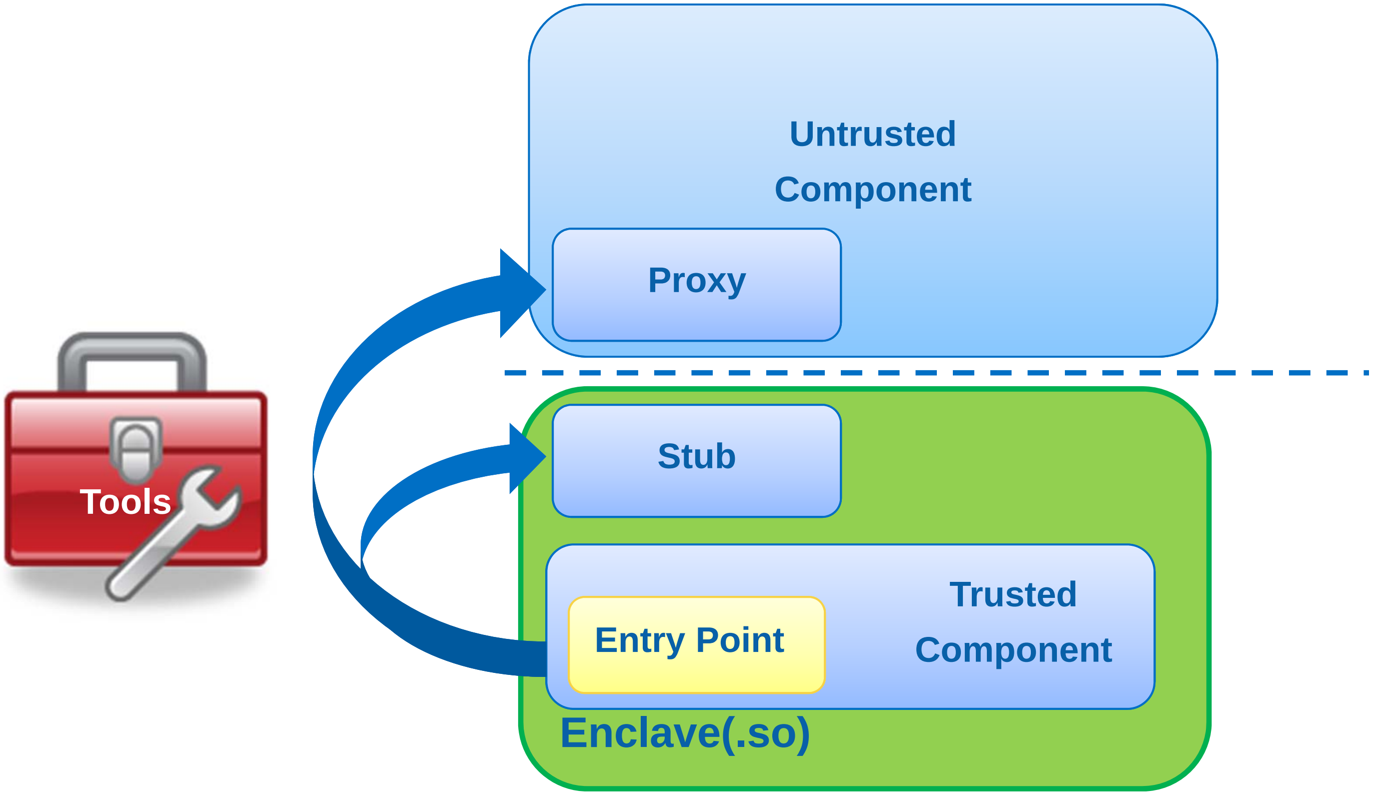


Hardening an Application with SGX



4. Identify Entry Points into Trusted Code

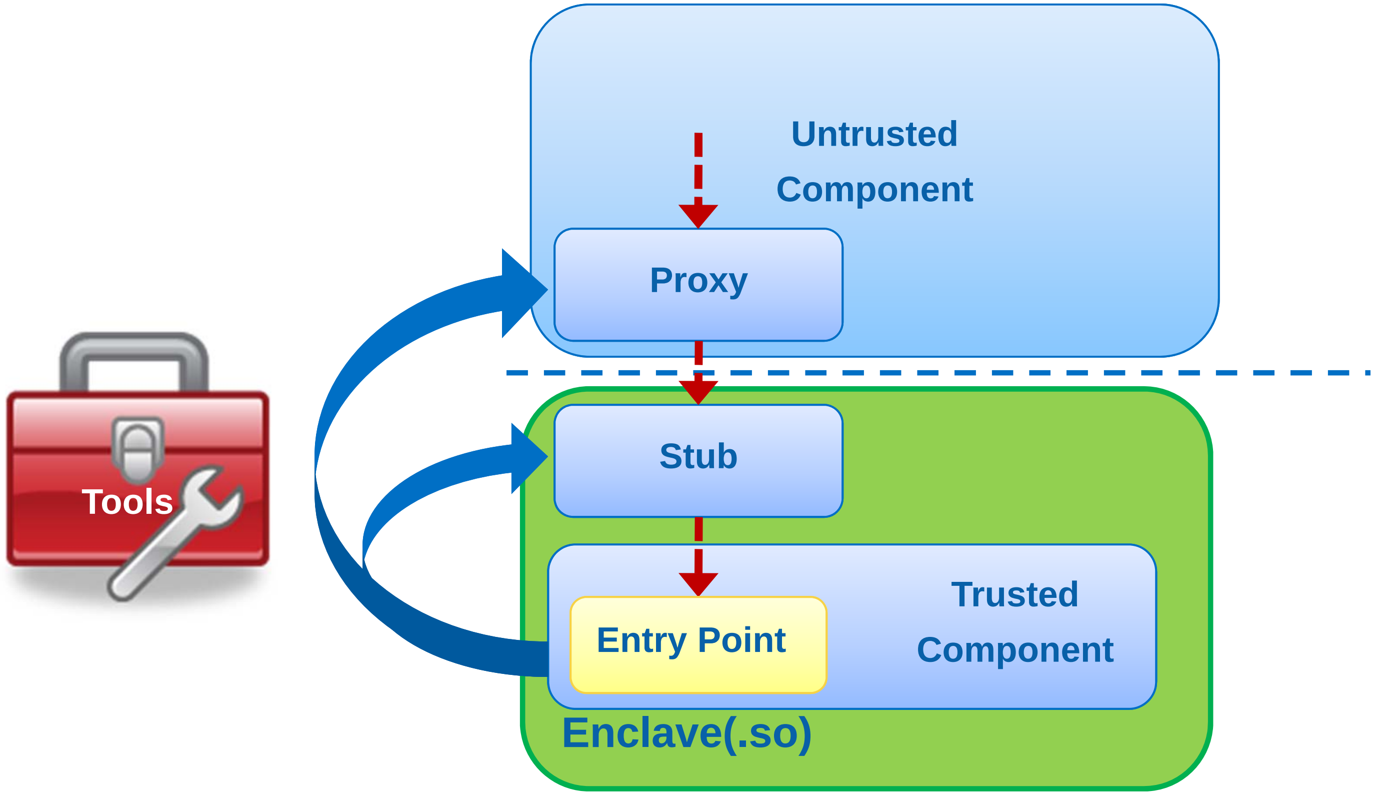
Hardening an Application with SGX



4. Identify Entry Points into Trusted Code

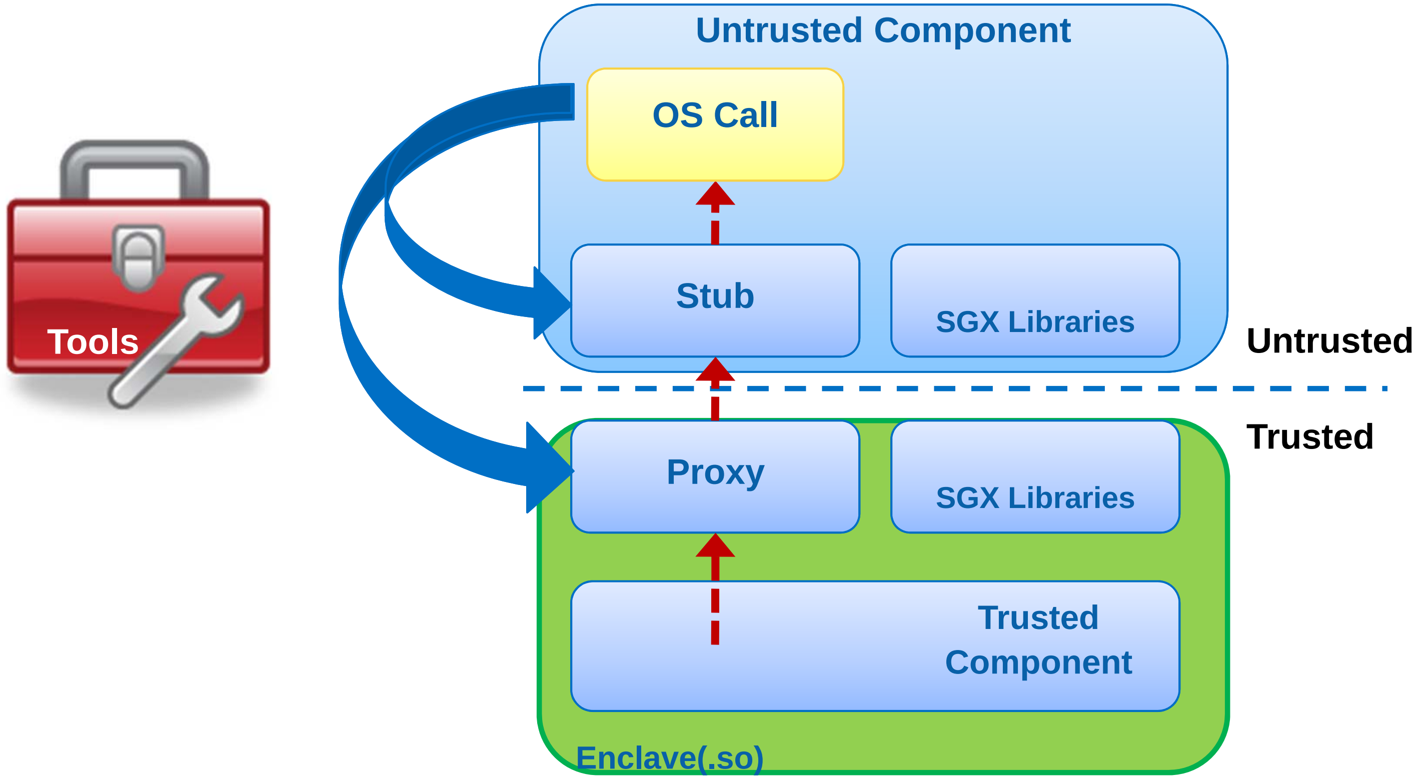
5. Use Tools to Create Glue Code

Hardening an Application with SGX



4. Identify Entry Points into Trusted Code
5. Use Tools to Create Glue Code
   - Untrusted component can now call into trusted component (ECall)

Hardening an Application with SGX



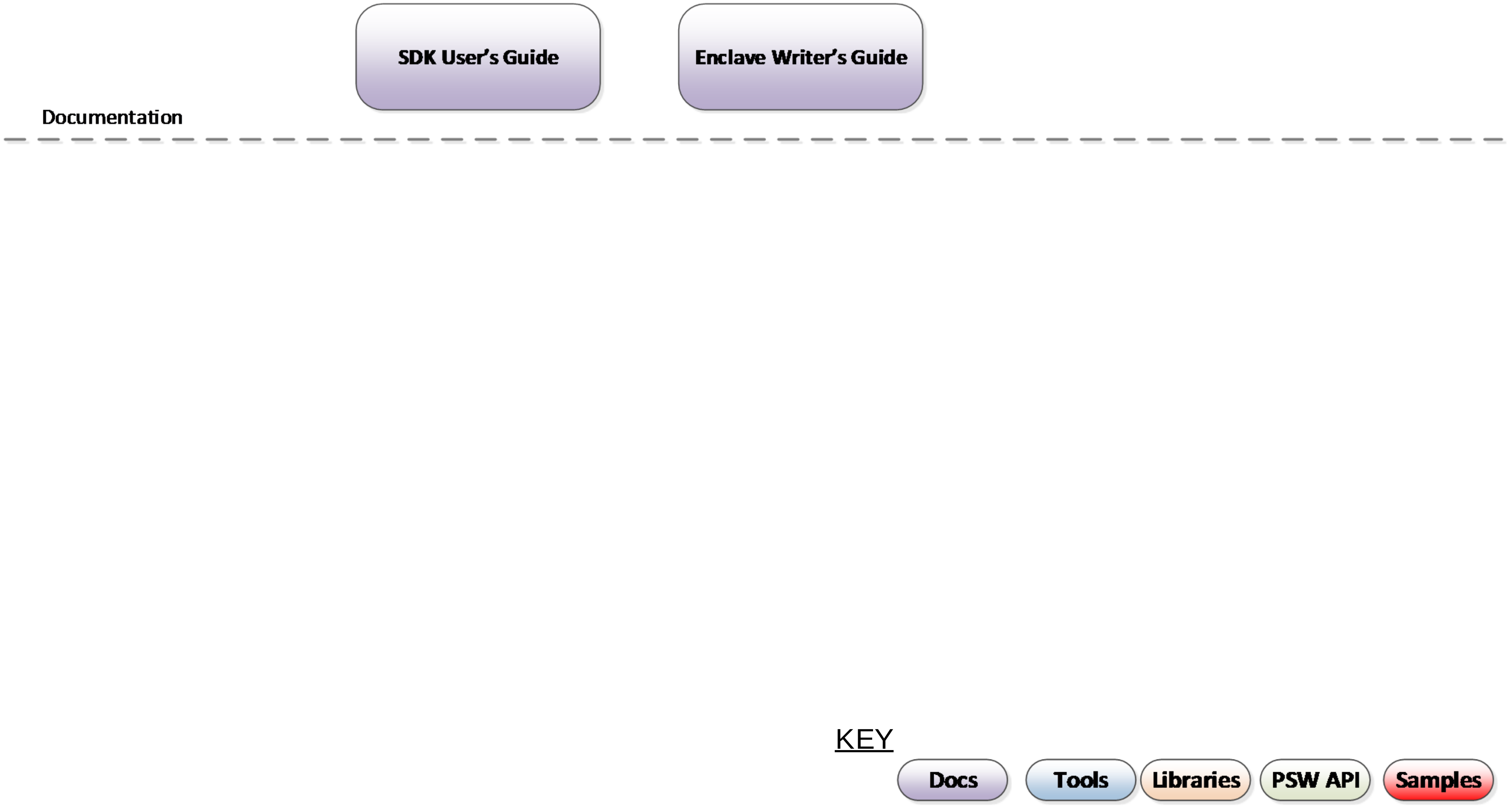
6. Need an OS Call from the Enclave?
   - Use Tools to Create Glue Code for Enclave calls to the Application (OCall)
7. Use SGX Libraries in the Application and the Enclave
   - Call loader API to Load the Enclave

Example SDK for Linux

KEY

Docs Tools Libraries PSW API Samples

Example SDK for Linux



SDK User's Guide

Enclave Writer's Guide

Documentation

KEY

Docs

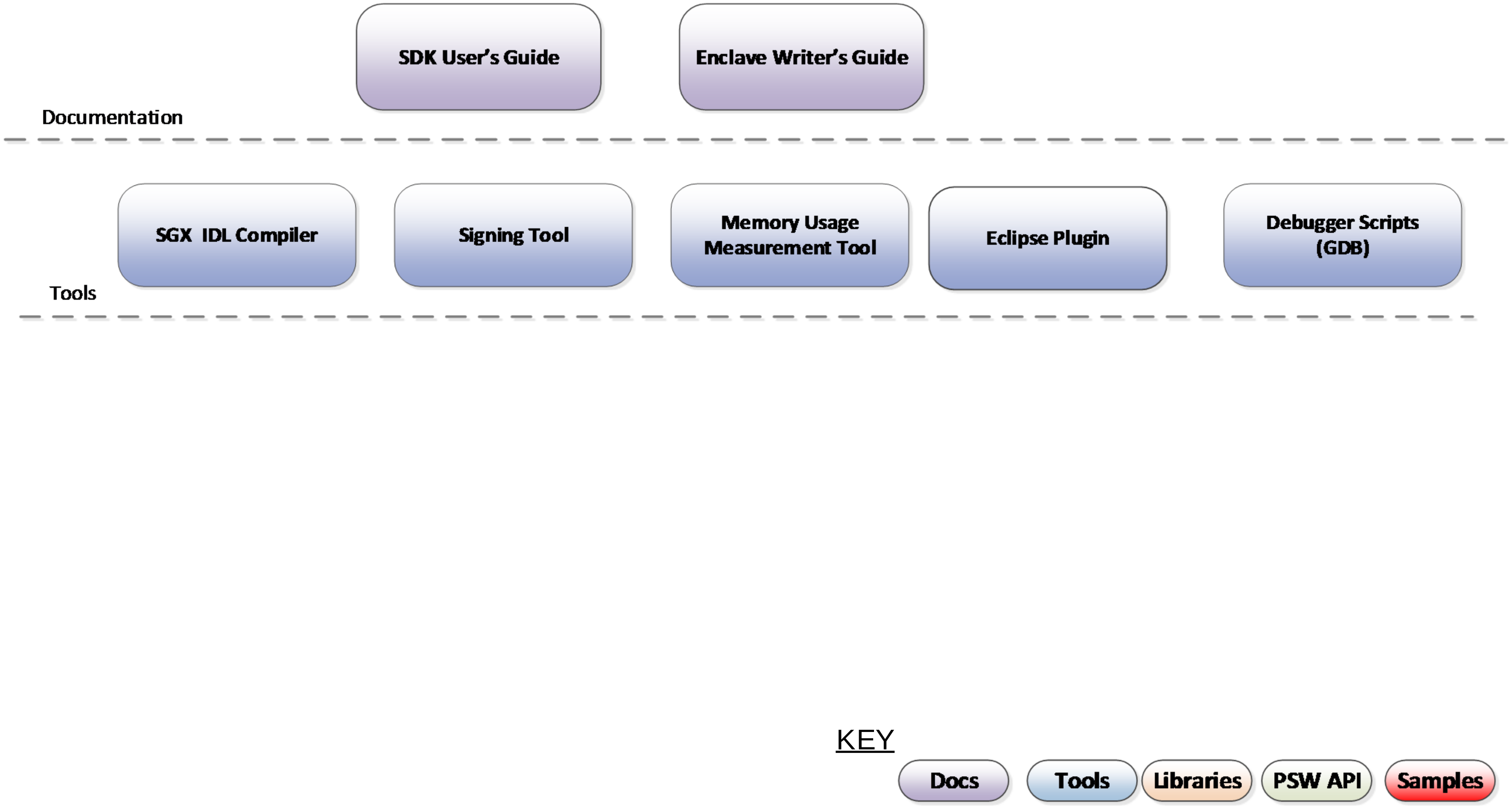
Tools

Libraries

PSW API

Samples

Example SDK for Linux



Documentation

SDK User's Guide

Enclave Writer's Guide

Tools

SGX IDL Compiler

Signing Tool

Memory Usage Measurement Tool

Eclipse Plugin

Debugger Scripts (GDB)

KEY

Docs

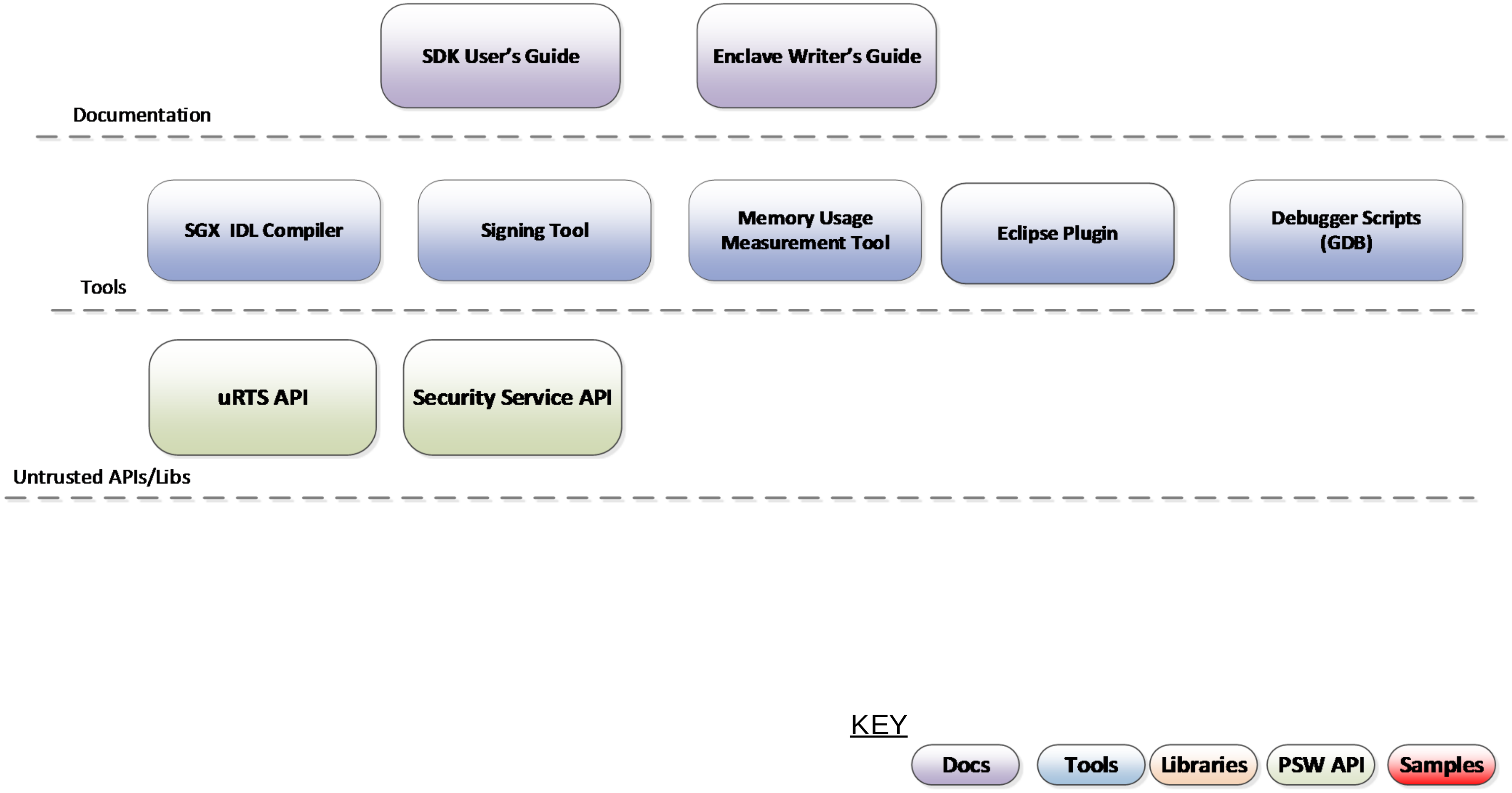
Tools

Libraries

PSW API

Samples

Example SDK for Linux



Documentation

SDK User's Guide

Enclave Writer's Guide

Tools

SGX IDL Compiler

Signing Tool

Memory Usage Measurement Tool

Eclipse Plugin

Debugger Scripts (GDB)

Untrusted APIs/Libs

uRTS API

Security Service API

KEY

Docs

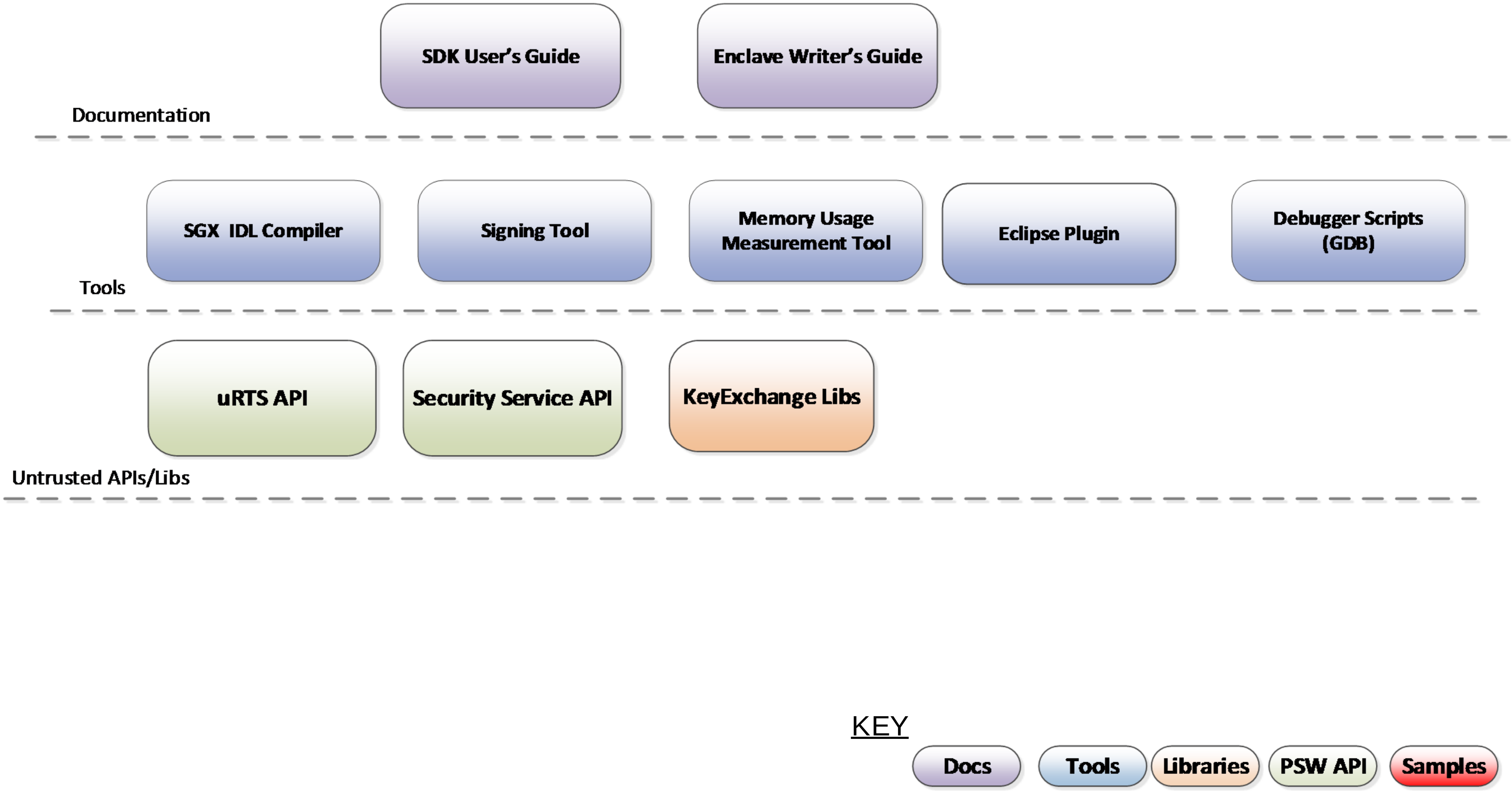
Tools

Libraries

PSW API

Samples

Example SDK for Linux



SDK User's Guide

Enclave Writer's Guide

Documentation

SGX IDL Compiler

Signing Tool

Memory Usage Measurement Tool

Eclipse Plugin

Debugger Scripts (GDB)

Tools

uRTS API

Security Service API

KeyExchange Libs

Untrusted APIs/Libs

KEY

Docs

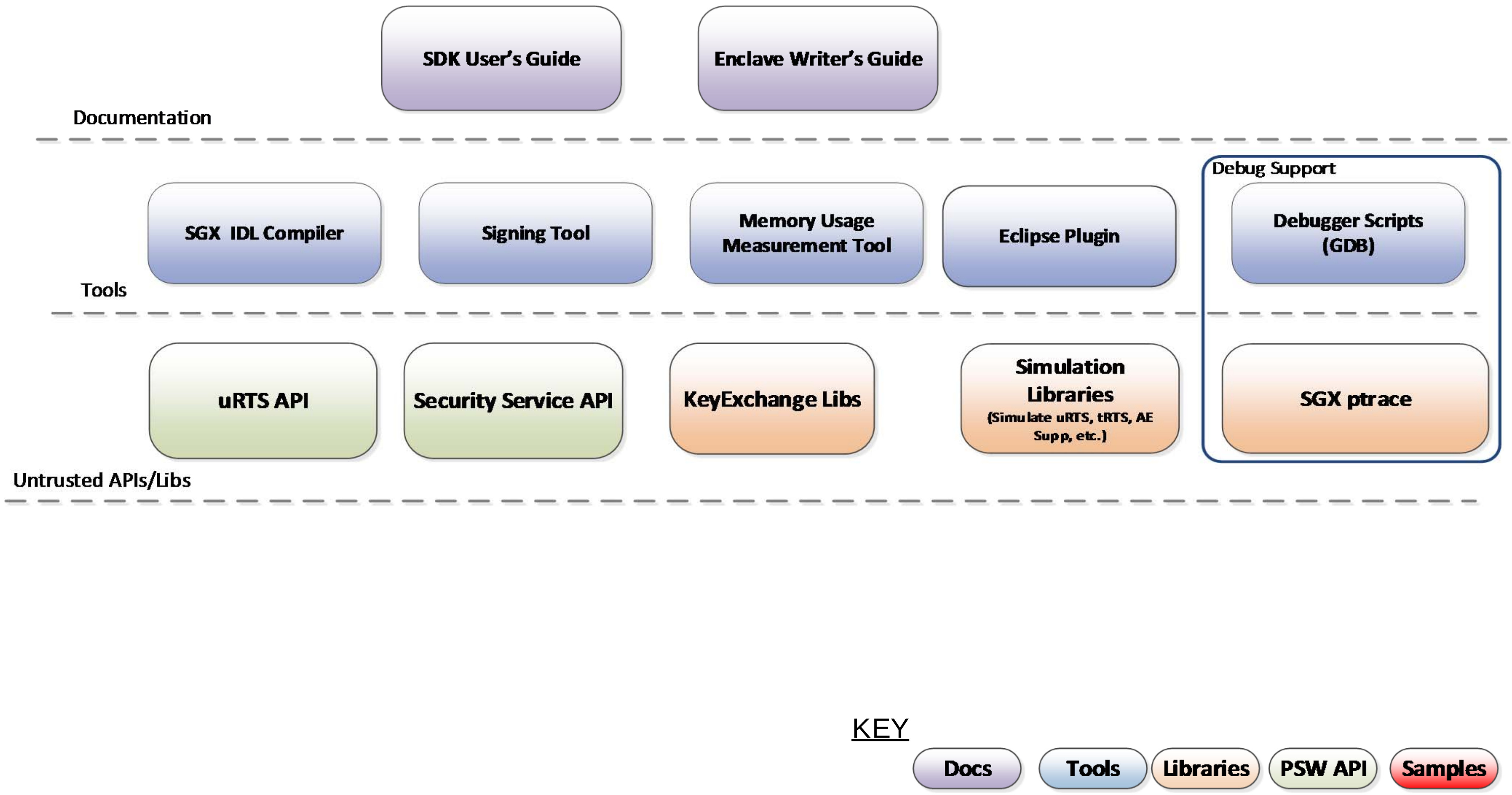
Tools

Libraries

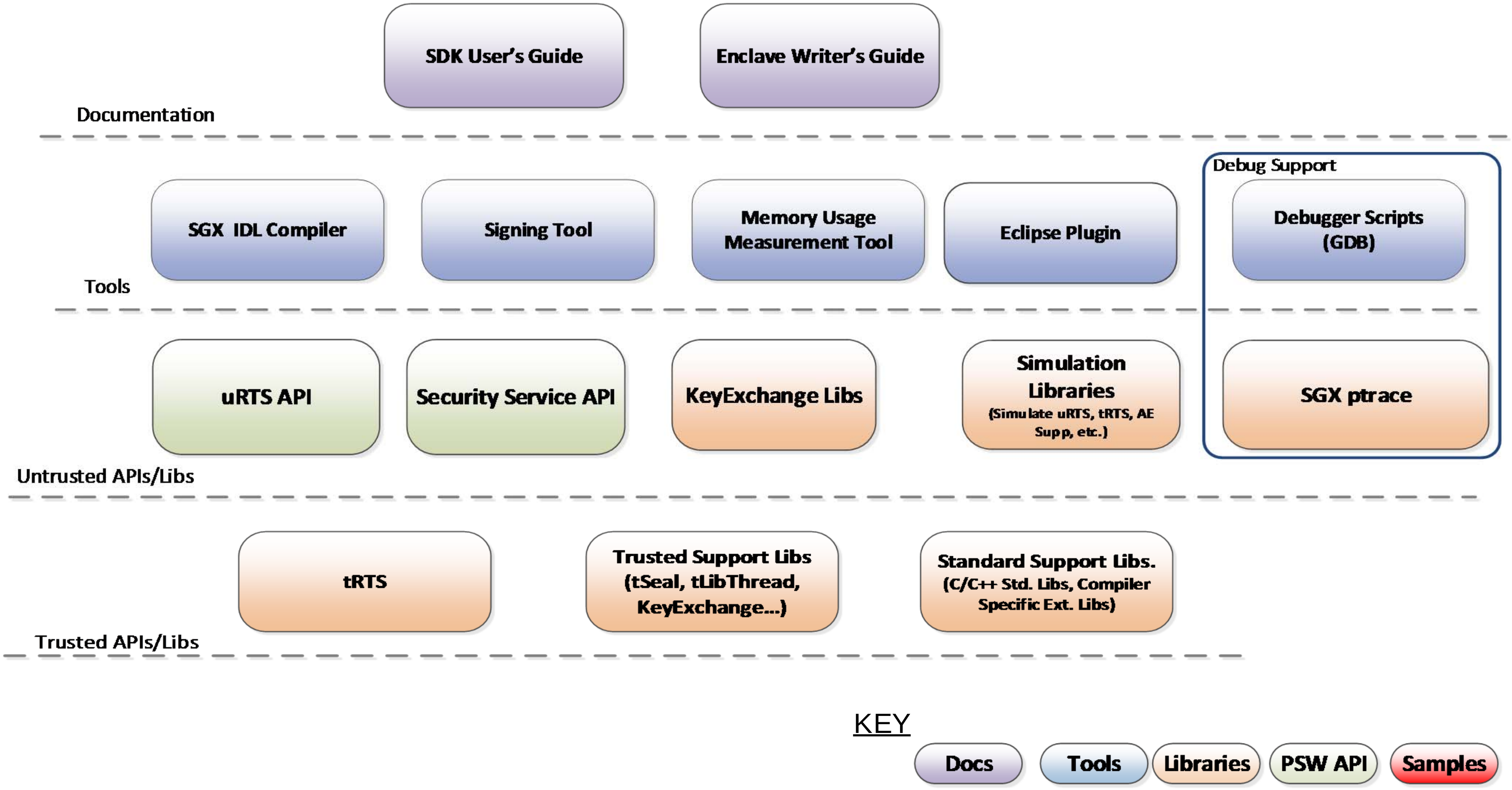
PSW API

Samples

Example SDK for Linux



Example SDK for Linux



Documentation

SDK User's Guide

Enclave Writer's Guide

Tools

SGX IDL Compiler

Signing Tool

Memory Usage Measurement Tool

Eclipse Plugin

Debug Support

Debugger Scripts (GDB)

Untrusted APIs/Libs

uRTS API

Security Service API

KeyExchange Libs

Simulation Libraries (Simulate uRTS, tRTS, AE Supp, etc.)

SGX ptrace

Trusted APIs/Libs

tRTS

Trusted Support Libs (tSeal, tLibThread, KeyExchange...)

Standard Support Libs. (C/C++ Std. Libs, Compiler Specific Ext. Libs)

KEY

Docs

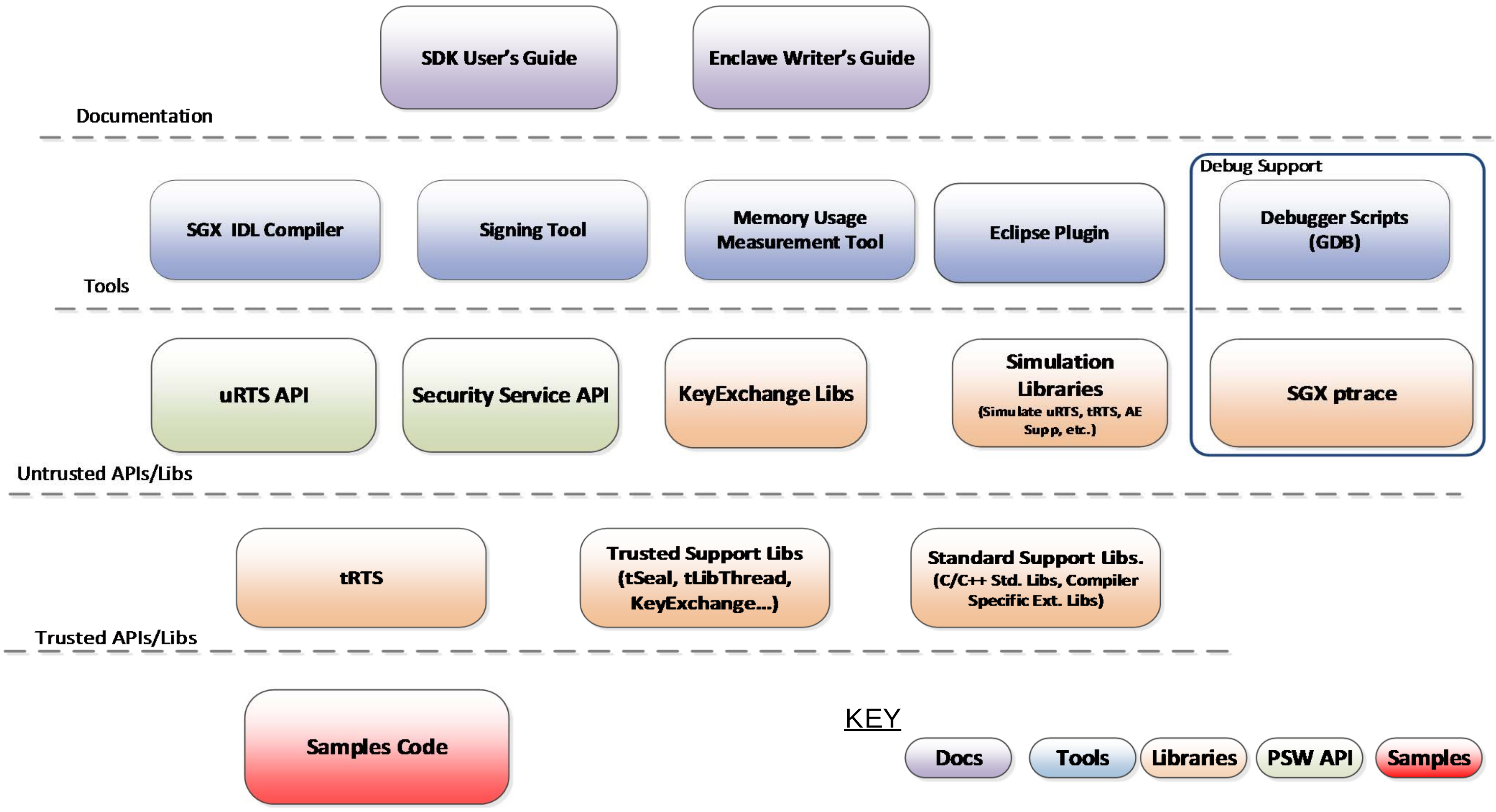
Tools

Libraries

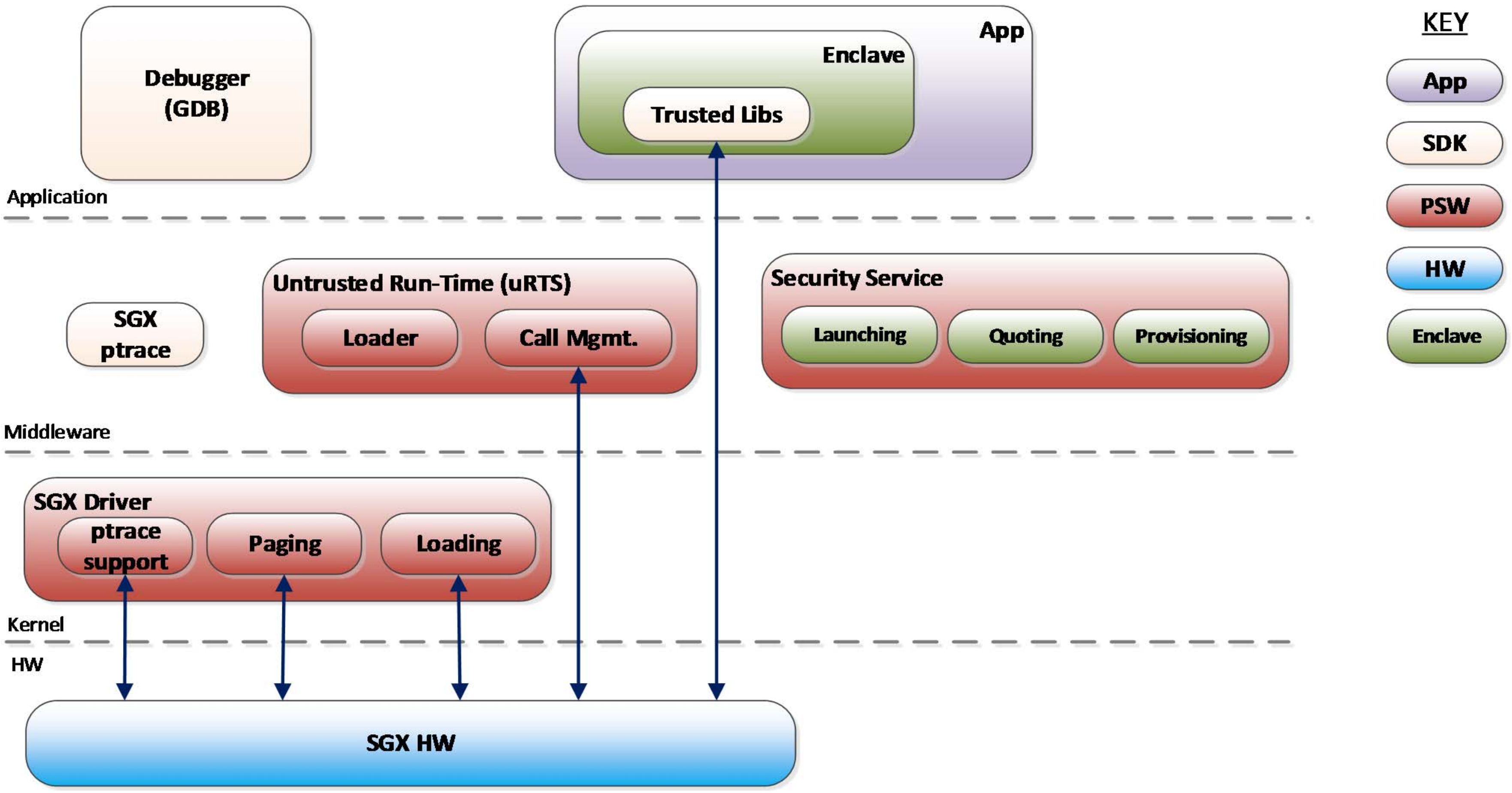
PSW API

Samples

Example SDK for Linux



Intel® SGX Platform SW



Summary

- Intel® SGX provides outstanding data protection and a simple programming model
  - An enclave limits the size of the TCB
  - Enclaves are protected in face of a compromised OS/VMM.
- Developers may focus on securing the smaller TCB
- Enclaves run within the application process
  - be built and debugged with familiar tools.
- The Intel SGX SW stack and tools should simplify development even more.

SGX Technical Summary

- Provides any application the ability to keep a secret
  - Provide capability using new processor instructions
  - Application can support multiple enclaves
- Provides integrity and confidentiality
  - Resists hardware attacks
  - Prevent software access, including privileged software and SMM
- Applications run within OS environment
  - Low learning curve for application developers
  - Open to all developers
- Resources managed by system software

Links

Joint research poster session: http://sigops.org/sosp/sosp13/

Public Cloud Paper using SGX2:

https://www.usenix.org/sites/default/files/osdi14_full_proceedings.pdf

Programming Reference for SGX1 & SGX2:
http://www.intel.com/software/isa

HASP Workshop:
https://sites.google.com/site/haspworkshop2013/workshop-program

ISCA 2015 Tutorial Link:
http://sgxisca.weebly.com/