A TALE OF TWO WORLDS

ASSESSING THE VULNERABILITY OF ENCLAVE SHIELDING RUNTIMES

PRODUCED BY KU LOUVEN AND BIRMINGHAM UNIVERSITIES
About imec-DistriNet enclave research  

- Trusted computing **across the system stack:** hardware, compiler, OS, application
- Integrated **attack-defense** perspective and **open-source** prototypes

CPU vulnerability research  
[VBMW\(^+\)18, SLM\(^+\)19, MOG\(^+\)20]

SGX-Step framework  
[VBPS17]

Sancus enclave processor  
[NVBM\(^+\)17]
Outline: How to besiege a fortress?

💡 Idea: security is weakest at the input/output interface(!)
Outline: How to besiege a TEE enclave?

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Tier1 (ABI)</th>
<th>Tier2 (API)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1 Entry status flags sanitization</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#2 Entry stack pointer restore</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#3 Exit register leakage</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#4 Missing pointer range check</td>
<td>⭐️</td>
<td>⭐️ ⭐️ ⭐️</td>
</tr>
<tr>
<td>#5 Null-terminated string handling</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#6 Integer overflow in range check</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#7 Incorrect pointer range check</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#8 Double fetch untrusted pointer</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#9 Ocall return value not checked</td>
<td>⭐️</td>
<td>⭐️ ⭐️ ⭐️</td>
</tr>
<tr>
<td>#10 Uninitialized padding leakage</td>
<td>⭐️</td>
<td>⭐️ ⭐️ ⭐️</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runtime</th>
<th>SGX-SDK</th>
<th>OpenEnclave</th>
<th>Graphene</th>
<th>SGX-LKL</th>
<th>Rust-EDP</th>
<th>Asylo</th>
<th>Keystone</th>
<th>Sancus</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1 Entry status flags sanitization</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#2 Entry stack pointer restore</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#3 Exit register leakage</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#4 Missing pointer range check</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#5 Null-terminated string handling</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#6 Integer overflow in range check</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#7 Incorrect pointer range check</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#8 Double fetch untrusted pointer</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#9 Ocall return value not checked</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
<tr>
<td>#10 Uninitialized padding leakage</td>
<td>[LK17]</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
<td>⭐️</td>
</tr>
</tbody>
</table>

Summary: > 35 enclave interface sanitization vulnerabilities across 8 projects
### Outline: How to besiege a TEE enclave?

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Runtime</th>
<th>SGX-SDK</th>
<th>OpenEnclave</th>
<th>Graphene</th>
<th>SGX-LKL</th>
<th>Rust-EDP</th>
<th>Asylo</th>
<th>Keystone</th>
<th>Sancus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tier 1 (ABI)</td>
<td></td>
<td></td>
<td>★</td>
<td>★</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td></td>
</tr>
<tr>
<td>#1 Entry status flags sanitization</td>
<td></td>
<td></td>
<td>★</td>
<td>★</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td></td>
</tr>
<tr>
<td>#2 Entry stack pointer restore</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
<td>⬤</td>
<td>⬤</td>
<td>●</td>
<td>○</td>
<td>⬤</td>
</tr>
<tr>
<td>#3 Exit register leakage</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
<td>⬤</td>
<td>⬤</td>
<td>●</td>
<td>○</td>
<td>⬤</td>
</tr>
<tr>
<td>Tier 2 (API)</td>
<td></td>
<td></td>
<td>★</td>
<td>★</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>★</td>
</tr>
<tr>
<td>#4 Missing pointer range check</td>
<td></td>
<td></td>
<td>★</td>
<td>★</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>★</td>
</tr>
<tr>
<td>#5 Null-terminated string handling</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
<td>⬤</td>
<td>⬤</td>
<td>●</td>
<td>○</td>
<td>★</td>
</tr>
<tr>
<td>#6 Integer overflow range check</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
<td>⬤</td>
<td>⬤</td>
<td>●</td>
<td>○</td>
<td>★</td>
</tr>
<tr>
<td>#7 Incorrect pointer range check</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
<td>⬤</td>
<td>⬤</td>
<td>●</td>
<td>○</td>
<td>★</td>
</tr>
<tr>
<td>#8 Double fetch untrusted pointer</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
<td>⬤</td>
<td>⬤</td>
<td>●</td>
<td>○</td>
<td>★</td>
</tr>
<tr>
<td>#9 Ocall return value not checked</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
<td>⬤</td>
<td>⬤</td>
<td>●</td>
<td>○</td>
<td>★</td>
</tr>
<tr>
<td>#10 Uninitialized padding leakage</td>
<td></td>
<td></td>
<td>○</td>
<td>○</td>
<td>⬤</td>
<td>⬤</td>
<td>●</td>
<td>○</td>
<td>★</td>
</tr>
</tbody>
</table>

**Impact:** 5 CVEs ... and lengthy embargo periods
Why do we need enclave fortresses anyway?
The big picture: Enclaved execution attack surface

Traditional **layered designs**: large trusted computing base
The big picture: Enclaved execution attack surface

Intel SGX promise: hardware-level isolation and attestation
The big picture: Enclaved execution attack surface

Previous attacks: exploit microarchitectural bugs or side-channels at the hardware level
The big picture: Enclaved execution attack surface

Idea: what about vulnerabilities in the trusted enclave software itself?
What do these projects have in common?
Why isolation is not enough: Enclave shielding runtimes

- TEE promise: enclave == “secure oasis” in a hostile environment
Why isolation is not enough: Enclave shielding runtimes

- TEE promise: enclave == “secure oasis” in a **hostile environment**
- ...but application writers and compilers are largely unaware of **isolation boundaries**
Why isolation is not enough: Enclave shielding runtimes

- TEE promise: enclave == “secure oasis” in a **hostile environment**
- ...but application writers and compilers are largely unaware of **isolation boundaries**

PAGE 4

----

Trusted **shielding runtime** transparently acts as a **secure bridge** on enclave entry/exit
...but what if the bridge itself is flawed?
Enclave shielding responsibilities

⚠️ Key questions: how to securely bootstrap from the untrusted world to the enclaved application binary (and back)? Which sanitizations to apply?
Key insight: split sanitization responsibilities across the ABI and API tiers: machine state vs. higher-level programming language interface
Tier 1: Establishing a trustworthy enclave ABI

Tier 1
ABI

Tier 2
API

Tier 3
APP
Tier1: Establishing a trustworthy enclave ABI

〜 Attacker controls CPU register contents on enclave entry/exit
leftrightarrow Compiler expects well-behaved calling convention (e.g., stack)
⇒ Need to initialize CPU registers on entry and scrub before exit!
Tier1: Establishing a trustworthy enclave ABI

↝ Attacker controls CPU register contents on enclave entry/exit

leftrightarrow Compiler expects well-behaved calling convention (e.g., stack)

⇒ Need to initialize CPU registers on entry and scrub before exit!

ABI vulnerability analysis

🔍 Relatively well-understood, but special care for stack pointer + status register
## Summary: ABI-level attack surface

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SGX-SDK</td>
</tr>
<tr>
<td>#1 Entry status flags sanitization</td>
<td>★</td>
</tr>
<tr>
<td>#2 Entry stack pointer restore</td>
<td>○</td>
</tr>
<tr>
<td>#3 Exit register leakage</td>
<td>○</td>
</tr>
</tbody>
</table>

Read the paper for several exploitable ABI vulnerabilities!
## Summary: ABI-level attack surface

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tier1 (ABI)</td>
<td></td>
</tr>
<tr>
<td>#1 Entry status flags sanitization</td>
<td>⭐ ⭐ 🌏 ⚫ ⚫</td>
</tr>
<tr>
<td>#2 Entry stack pointer restore</td>
<td>⚫ ⚫ ⚫ ⚫ ⚫</td>
</tr>
<tr>
<td>#3 Exit register leakage</td>
<td>⚫ ⚫ ⚫ ⚫ ⚫</td>
</tr>
</tbody>
</table>

### Attack surface

- **Complex x86 ABI (Intel SGX) >> simpler RISC designs**

---

*A lesson on complexity*
x86 string instructions: Direction Flag (DF) operation

- Special x86 rep string instructions to speed up streamed memory operations

```c
/* memset(buf, 0x0, 100) */
for (int i=0; i < 100; i++)
    buf[i] = 0x0;
```

```assembly
1 lea rdi, buf
2 mov al, 0x0
3 mov ecx, 100
4 rep stos [rdi], al
```
x86 string instructions: Direction Flag (DF) operation

- Special x86 rep string instructions to speed up streamed memory operations
- Default operate left-to-right

```c
/* memset(buf, 0x0, 100) */
for (int i=0; i < 100; i++)
buf[i] = 0x0;
```

```assembly
1  lea rdi, buf
2  mov al, 0x0
3  mov ecx, 100
4  rep stos [rdi], al
```
x86 string instructions: Direction Flag (DF) operation

- Special x86 rep string instructions to speed up streamed memory operations
- Default operate left-to-right, unless software sets RFLAGS.DF=1

```c
/* memset(buf, 0x0, 100) */
for (int i=0; i < 100; i++)
    buf[i] = 0x0;
```

```
lea rdi, buf+100
mov al, 0x0
mov ecx, 100
std; set direction flag
rep stos [rdi], al
```
SGX-DF: Inverting enclaved string memory operations

x86 System-V ABI

8 The direction flag DF in the %eFLAGS register must be clear (set to “forward” direction) on function entry and return. Other user flags have no specified role in the standard calling sequence and are not preserved across calls.
Enter enclave with \textit{RFLAGS.DF=0}
Intended heap memory initialization: left-to-right

enclave_func:

buf = malloc(100);
memset(buf, 0x00, 100);

enclave_heap:
Enter enclave with \texttt{RFLAGS.DF=1}

\begin{verbatim}
buf = malloc(100);
memset(buf, 0x00, 100);
\end{verbatim}
Enclave heap memory corruption: right-to-left...

**enclave_func:**

```c
buf = malloc(100);
memset(buf, 0x00, 100);
```

**enclave_heap:**
Summary:
A potential security vulnerability in Intel SGX SDK may allow for information disclosure, escalation of privileges, or denial of service. Intel is releasing software updates to mitigate this potential vulnerability. This potential vulnerability is present in all SGX-enabled builds with the affected SGX SDK versions.

Vulnerability Details:
CVE ID: CVE-2019-14568
Description: Insufficient input validation in Intel SGX SDK versions shown below may allow an authenticated user to enable information disclosure, escalation of privilege, or denial of service via local access.
CVSS Base Score: 7.0 (High)

CVE ID: CVE-2019-14563
Description: Insufficient input validation in Intel SGX SDK versions shown below may allow an authenticated user to enable information disclosure, escalation of privilege, or denial of service via local access.
CVSS Base Score: 7.0 (High)
There’s more! **Alignment Check (AC) flag** enables **exceptions** for **unaligned data accesses** → **intra-cacheline side-channel** 😊

```c
uint16_t d = lookup_table[secret];
```

**enclave_func:**

**enclave_data:**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
</table>

64B cacheline
Enter enclave with \textit{RFLAGS.AC=1} and secret index=0 → well-aligned data access: \textit{no exception}
SGX-AC: Building an intra-cacheline side-channel

Enter enclave with $RFLAGS.AC=1$ and secret index=1
→ unaligned data access: alignment-check exception...
Tier 2: Sanitizing the enclave API

Tier 1
ABI

Tier 2
API

Tier 3
APP
Validating pointer arguments: Confused deputy attacks

```
main (argv[0]=&hello)

"hello, world"
```

```
$ "hello, world"
```
Validating pointer arguments: Confused deputy attacks

- \texttt{main} (argv[0]=&\texttt{secret})
- enclave memory
- untrusted memory
- $ "top secret!"$
Validating pointer arguments: Confused deputy attacks
Idea: 2-stage approach ensures string arguments fall *entirely* outside enclave

```c
int my_ecall(char *s)
{
    len = strlen(s);
    if (!outside_enclave(s, len))
        return ILLEGAL_ARG;
    ... return SUCCESS;
}
char *arg = "hello, world";
```
...but what if we try passing an illegal, in-enclave pointer anyway?

```c
int my_ecall(char *s) {
    len = strlen(s);
    if (!outside_enclave(s, len))
        return ILLEGAL_ARG;
    ... return SUCCESS;

bool secret1 = 1;
bool secret2 = 0;
```
Intel SGX-SDK: Null-terminated strings are hard...  

⚠️ Enclave **first** computes **length of secret, in-enclave buffer!**

```c
int my_ecall(char *s)
{
    len = strlen(s);
    if (!outside_enclave(s, len))
        return ILLEGAL_ARG;
    ...
    return SUCCESS;
}
```

```c
bool secret1 = 1;
bool secret2 = 0;
```
... and only **afterwards verifies** whether *entire string* falls outside enclave.

```c
int my_ecall(char *s)
{
    len = strlen(s);
    if (!outside_enclave(s, len))
        return ILLEGAL_ARG;
    ...
    return SUCCESS;
}

bool secret1 = 1;
bool secret2 = 0;
```
Idea: `strlen()` timing as a side-channel oracle for in-enclave null bytes 😊
Challenge: Building a precise null byte oracle

What about measuring execution time?
Building the oracle with `strlen()` timing?

**Execution timing side-channel?**

⚠️ **Too noisy:** we need to measure timing of a single x86 increment instruction...
Challenge: Building a precise null byte oracle

What about measuring page faults?
Protection from Side-Channel Attacks

Intel® SGX does not provide explicit protection from side-channel attacks. It is the enclave developer's responsibility to address side-channel attack concerns.

In general, enclave operations that require an OCall, such as thread synchronization, IO, etc., are exposed to the untrusted domain. If using an OCall would allow an attacker to gain insight into enclave secrets, then there would be a security concern. This scenario would be classified as a side-channel attack, and it would be up to the ISV to design the enclave in a way that prevents the leaking of side-channel information.

An attacker with access to the platform can see what pages are being executed or accessed. This side-channel vulnerability can be mitigated by aligning specific code and data blocks to exist entirely within a single page.

More important, the application enclave should use an appropriate crypto implementation that is side channel attack resistant inside the enclave if side-channel attacks are a concern.

https://software.intel.com/en-us/node/703016
Counting `strlen()` loop iterations with page faults?

**Temporal resolution:** progress requires both code + data pages mapped in.
Challenge: Counting `strlen()` loop iterations

What about leveraging interrupts?
SGX-Step: Executing enclaves one instruction at a time


https://github.com/jovanbulck/sgx-step
SGX-Step: Executing enclaves one instruction at a time

1. IRQ
2. AEX
3. IRQ Handler
4. IRET
5. AEP
6. ERESUME

if secret do
    inst1
else
    inst2
endif

rdtsc
mov %eax, (tsc2)
iretq

... movl $TMR, 0xceee00380
rdtsc
mov %eax, (tsc1)
mov $ERESUME, %rax
enclu

/dev/sgx step

https://github.com/jovanbulck/sgx-step
Building a deterministic `strlen()` null byte oracle with SGX-Step

Building a deterministic `strlen()` null byte oracle with SGX-Step

CVE-2018-3626: ALL YOUR ZERO BYTES ARE BELONG TO US
Breaking AES-NI with the `strlen()` null byte oracle
Breaking AES-NI with the `strlen()` null byte oracle

```
aesenc xmm0
aesenc xmm0
aesenclast xmm0

Enclave SSA memory
```

```
0xAB 0x82 0x99 0x00
```

Ciphertext last round

```
AB 82 99 00 ...
```

```
Sbox

rk_{10} = Sbox(0) \oplus 0x3F
```

Ciphertext final

```
... ... ... ... ...

3F
```

INTERRUPT

strlen() oracle
Breaking AES-NI with the `strlen()` null byte oracle
## Summary: API-level attack surface

<table>
<thead>
<tr>
<th>Tier2 (API)</th>
<th>SGX-SDK</th>
<th>OpenEnclave</th>
<th>Graphene</th>
<th>SGX-LKL</th>
<th>Rust-EDP</th>
<th>Asylo</th>
<th>Keystone</th>
<th>Sancus</th>
</tr>
</thead>
<tbody>
<tr>
<td>#4 Missing pointer range check</td>
<td>○</td>
<td>★</td>
<td>★</td>
<td>★</td>
<td>○</td>
<td>★</td>
<td>○</td>
<td>★</td>
</tr>
<tr>
<td>#5 Null-terminated string handling</td>
<td>★</td>
<td>★</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>#6 Integer overflow in range check</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>#7 Incorrect pointer range check</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>#8 Double fetch untrusted pointer</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>#9 Ocall return value not checked</td>
<td>○</td>
<td>★</td>
<td>★</td>
<td>★</td>
<td>○</td>
<td>●</td>
<td>★</td>
<td>○</td>
</tr>
<tr>
<td>#10 Uninitialized padding leakage</td>
<td>[LK17]</td>
<td>★</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>★</td>
<td>★</td>
</tr>
</tbody>
</table>

Read the paper for more API attacks!
Summary: API-level attack surface

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Runtime</th>
<th>SGX-SDK</th>
<th>OpenEnclave</th>
<th>Graphene</th>
<th>SGX-LKL</th>
<th>Rust-EDP</th>
<th>Asylo</th>
<th>Keystone</th>
<th>Sancus</th>
</tr>
</thead>
<tbody>
<tr>
<td>#4 Missing pointer range check</td>
<td>⃝ ⃝ ⃝ ⃝</td>
<td>⃝ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
</tr>
<tr>
<td>#5 Null-terminated string handling</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
</tr>
<tr>
<td>#6 Integer overflow in range check</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
</tr>
<tr>
<td>#7 Incorrect pointer range check</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
</tr>
<tr>
<td>#8 Double fetch untrusted pointer</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
</tr>
<tr>
<td>#9 Ocall return value not checked</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
</tr>
<tr>
<td>#10 Uninitialized padding leakage [LK17]</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
<td>⚫ ⚫ ⚫ ⚫</td>
</tr>
</tbody>
</table>

⚠️ **Critical oversights** in production and research code

→ across TEEs and programming languages (incl. safe langs like Rust)
## Summary: API-level attack surface

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SGX-SDK</td>
</tr>
<tr>
<td>#4 Missing pointer range check</td>
<td>○</td>
</tr>
<tr>
<td>#5 Null-terminated string handling</td>
<td>✭</td>
</tr>
<tr>
<td>#6 Integer overflow in range check</td>
<td>○</td>
</tr>
<tr>
<td>#7 Incorrect pointer range check</td>
<td>○</td>
</tr>
<tr>
<td>#8 Double fetch untrusted pointer</td>
<td>○</td>
</tr>
<tr>
<td>#9 Ocall return value not checked</td>
<td>○</td>
</tr>
<tr>
<td>#10 Uninitialized padding leakage</td>
<td>[LK17]</td>
</tr>
</tbody>
</table>

- Generally understood (Iago attacks) but **still widespread**, not exclusive to library OSs

---

Checkoway et al. “Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface” ASPLOS 2013 [CS13]
Washes away Bacteria
Frequent hand washing helps keep your family healthy.

Safeguard®
White with touch of Aloe
Conclusions and outlook

Take-away message

🛡️ Secure enclave interactions require proper **ABI and API sanitizations**!
Conclusions and outlook

Take-away message

🛡️ Secure enclave interactions require proper **ABI and API sanitizations**!

- Large **attack surface**, including subtle **side-channel oversights**...
- **Defenses**: need to research more **principled sanitization strategies**
- **User-to-kernel analogy**: learn from experience with **secure OS development**

🌐 [https://github.com/jovanbulck/0xbadc0de](https://github.com/jovanbulck/0xbadc0de)
A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes

Jo Van Bulck¹  David Oswald²  Eduard Marin²  Abdulla Aldoseri²
Flavio D. Garcia²  Frank Piessens¹

¹imec-DistriNet, KU Leuven  ²The University of Birmingham, UK

Hardware-aided trusted computing devroom, FOSDEM, February 1, 2020
S. Checkoway and H. Shacham.
Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface.

S. Lee and T. Kim.
Leaking uninitialized secure enclave memory via structure padding.

Plundervolt: Software-based fault injection attacks against intel sgx.

Sancus 2.0: A low-cost security architecture for IoT devices.

ZombieLoad: Cross-privilege-boundary data sampling.
In *CCS*, 2019.

Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution.

J. Van Bulck, F. Piessens, and R. Strackx.
SGX-Step: A practical attack framework for precise enclave execution control.
J. Van Bulck, F. Piessens, and R. Strackx.
Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic.

Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution.
TEE design: Single-address-space vs. world-shared memory approaches
edger8r: Input/output buffer cloning

1. Trusted runtime
2. Cloned buffer (trusted memory)
3. Edger8r bridge
4. Application
Intel SGX `strlen` oracle attack

```c
encryptString(){
    aesenc k[8], %xmm0
    aesenc k[9], %xmm0
    // Interruption
    aesenclast k[10],%xmm0
}
```

1. Ecall (message)
2. AEX Thread A
3. Config timer
4. Ecall (SSA_frame + XMM0_OFFSET)
5. AEX Thread B
6. Check accessed bit

Thread A
- Ecall (msg)
- `strlen(msg)`

Thread B
- Encrypt string

Host Application
- `aesenc` function calls

Enclave
- `encryptString` function

Host Application
- `strlen` function

Hardware
- Timer configuration

Enclave
- Interrupt handling
Exploitation challenges: Building a precise null byte oracle

Goal: Precisely measure `strlen()` loop iterations?

```c
size_t strlen(char *str) {
    char *s;
    for (s = str; *s; ++s);
    return (s - str);
}
```

⇒ tight loop: 4 asm instructions, single memory operand, single code + data page
Reconstructing the full AES-NI round key

**Algorithm 1** strlen() oracle AES key recovery where $S(\cdot)$ denotes the AES SBox and $SR(p)$ the position of byte $p$ after AES ShiftRows.

```plaintext
while not full key $K$ recovered do
    ($P, C, L$) ← random plaintext, associated ciphertext, strlen oracle
    if $L < 16$ then
        $K[SR(L)] ← C[SR(L)] \oplus S(0)$
    end if
end while
```