Finanziato dall'Unione europea
NextGenerationEU

Ministero dell'Università e della Ricerca

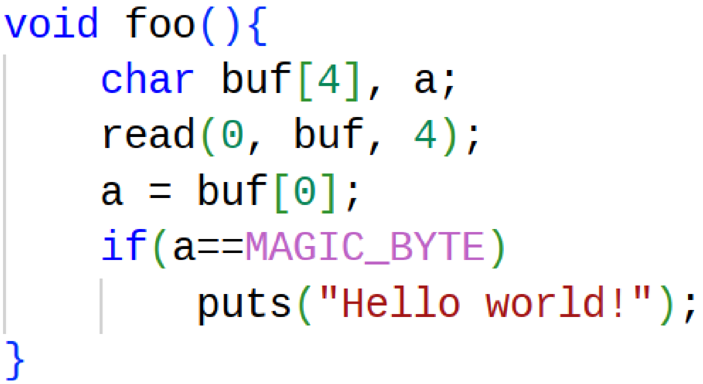
Italiadomani
PIANO NAZIONALE DI RIPRESA E RESILIENZA

SERICS
SECURITY AND RIGHTS IN THE CYBERSPACE

QMSan: Efficiently Detecting Uninitialized Memory Errors During Fuzzing

Matteo Marini (Sapienza)
Daniele Cono D’Elia (Sapienza)
Mathias Payer (EPFL)
Leonardo Querzoni (Sapienza)

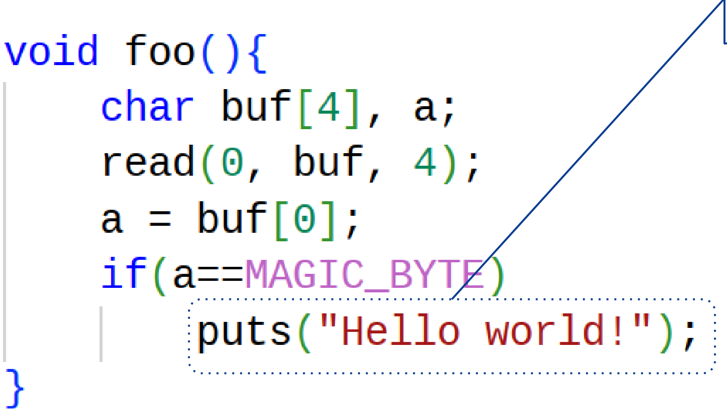
UUM Errors



```c
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```

UUM Errors

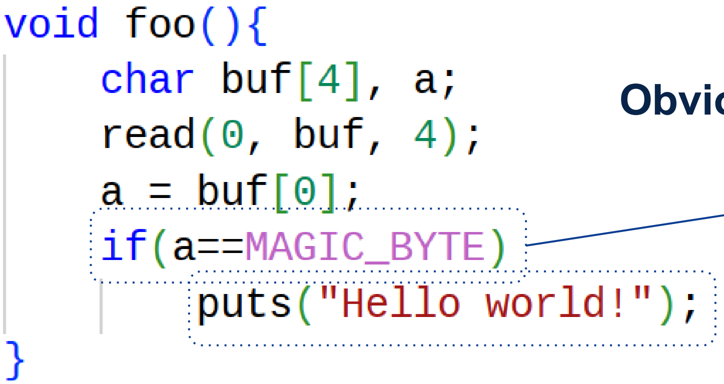
- Will this program print “*Hello world!*”?



```c
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```

UUM Errors

- Will this program print “*Hello world!*”?

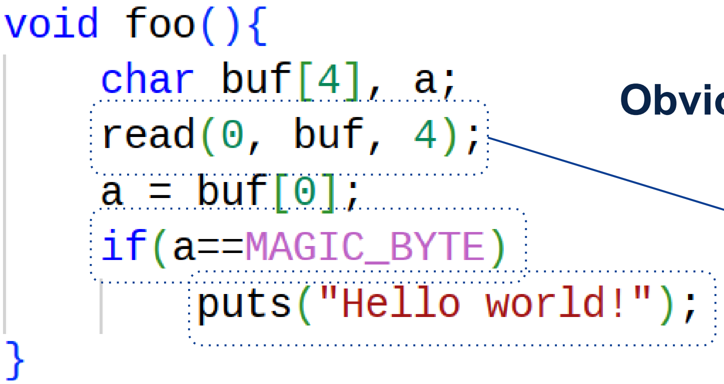


```
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```

Obvious answer: it depends on the first char of the buffer!

UUM Errors

- Will this program print “*Hello world!*”?

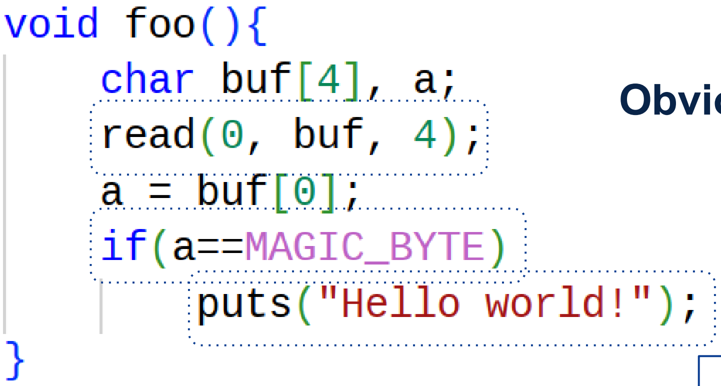


```
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```

Obvious answer: it depends on the first char of the buffer!

But what if **nothing is read**?

UUM Errors



```
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```

- Will this program print “*Hello world!*”?

Obvious answer: it depends on the first char of the buffer!

But what if **nothing is read**?

Use-of-Uninitialized-Memory (UUM) error!

UUM Errors - Detection

UUM Errors - Detection

- Define a **shadow memory**
    - Contains Initialization status of memory

UUM Errors - Detection

- Define a **shadow memory**
  - Contains Initialization status of memory

- **Propagate** the shadow memory
  - Propagation rules

UUM Errors - Detection

- Define a **shadow memory**
  - Contains Initialization status of memory

- **Propagate** the shadow memory
  - Propagation rules

- Check shadow memory on **memory usages**
  - Pointer derefentiation
  - Conditional branches
  - Data in system calls

UUM Errors - Detection

- Define a **shadow memory**
  - Contains Initialization status of memory

- **Propagate** the shadow memory
  - Propagation rules

- Check shadow memory on **memory usages**
  - Pointer derefentiation
  - Conditional branches
  - Data in system calls

?

UUM Errors - Detection

- Define a **shadow memory**
  - Contains Initialization status of memory

- **Propagate** the shadow memory
  - Propagation rules

- Check shadow memory on **memory usages**
  - Pointer derefentiation
  - Conditional branches
  - Data in system calls

?

Loading uninitialized data **is allowed…**

UUM Errors - Detection

- Define a **shadow memory**
  - Contains Initialization status of memory

- **Propagate** the shadow memory
  - Propagation rules

- Check shadow memory on **memory usages**
  - Pointer derefentiation
  - Conditional branches
  - Data in system calls

?

Loading uninitialized data **is allowed…**

…As long as **<u>its content is not used</u>**

Memory Sanitizer (MSan)

- State-of-the-Art UUM detection
  - Compile-time solution

Memory Sanitizer (MSan)

- State-of-the-Art UUM detection
  - Compile-time solution

Pros:

- Fast (2-3x slowdown)
- Accurate
- Fuzzing-compatible

Memory Sanitizer (MSan)

- State-of-the-Art UUM detection
  - Compile-time solution

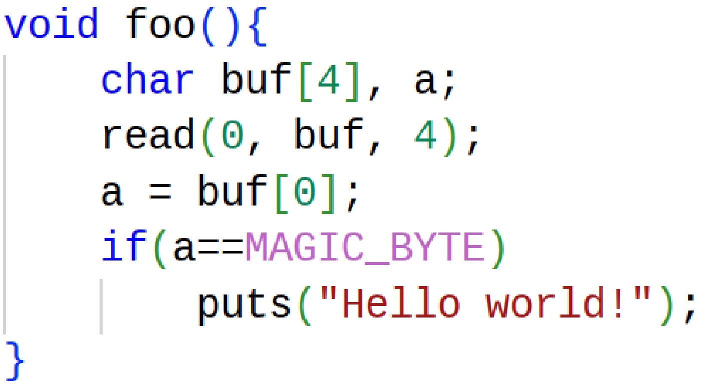
Pros:

- Fast (2-3x slowdown)
- Accurate
- Fuzzing-compatible

Cons:

- Requires **recompilation**
- **All code** must be instrumented
  - Libraries
- LLVM only

MSan - Workflow



```
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```

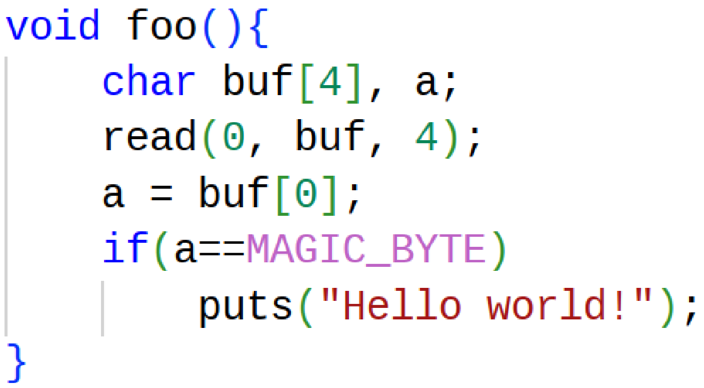
MSan - Workflow

Memory

? Unknown

Shadow memory

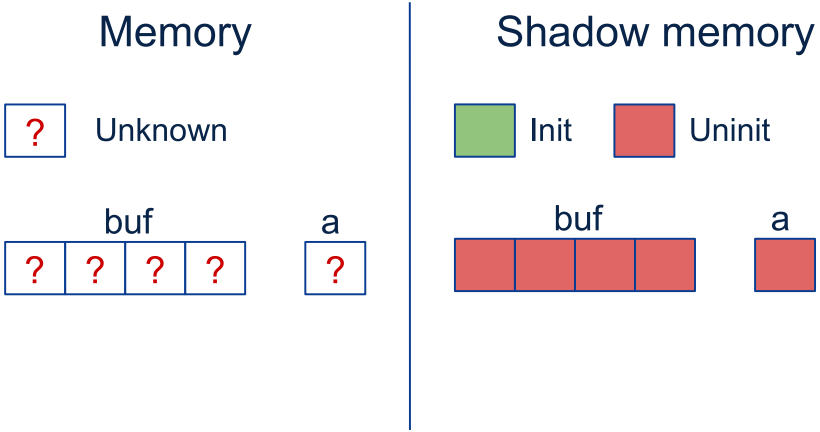
🟩 Init 🟥 Uninit



```
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```

MSan - Workflow

```
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```



Memory

? Unknown

buf

?	?	?	?

a

?

Shadow memory

🟩 Init 🟥 Uninit

buf

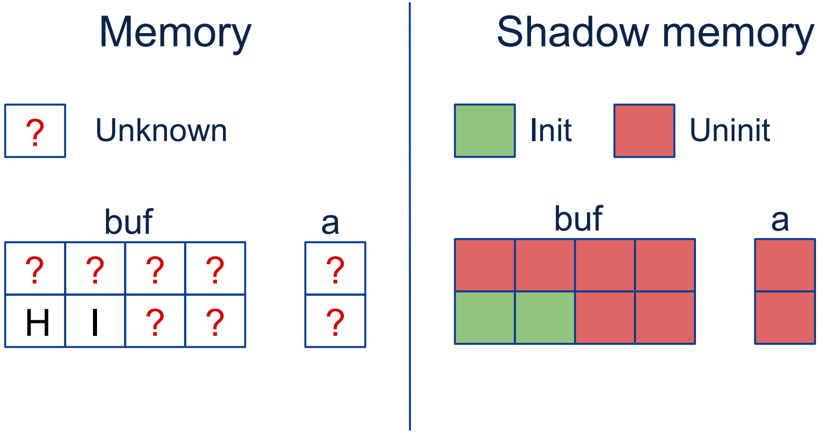
🟥	🟥	🟥	🟥

a

🟥

MSan - Workflow

```
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```



Memory

? Unknown

buf				a
?	?	?	?	?
H	I	?	?	?

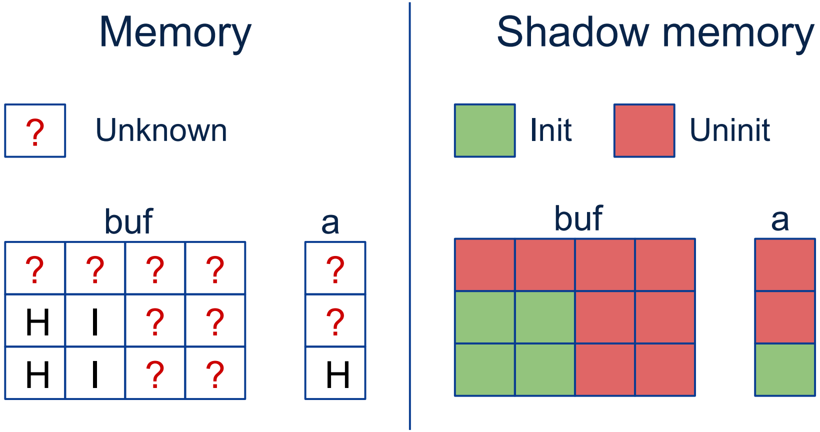
Shadow memory

🟩 Init 🟥 Uninit

buf				a
🟥	🟥	🟥	🟥	🟥
🟩	🟩	🟥	🟥	🟥

MSan - Workflow

```
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```



Memory

? Unknown

buf				a
?	?	?	?	?
H	I	?	?	?
H	I	?	?	H

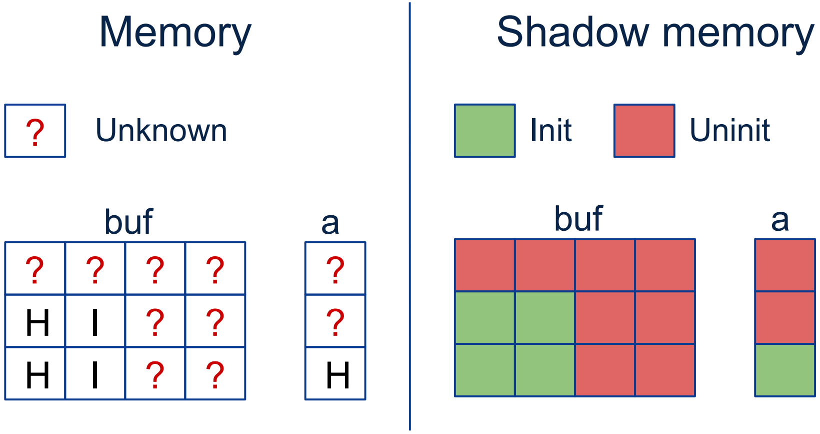
Shadow memory

🟩 Init 🟥 Uninit

buf				a
🟥	🟥	🟥	🟥	🟥
🟩	🟩	🟥	🟥	🟥
🟩	🟩	🟥	🟥	🟩

MSan - Workflow

```
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```



Memory

? Unknown

buf				a
?	?	?	?	?
H	I	?	?	?
H	I	?	?	H

Shadow memory

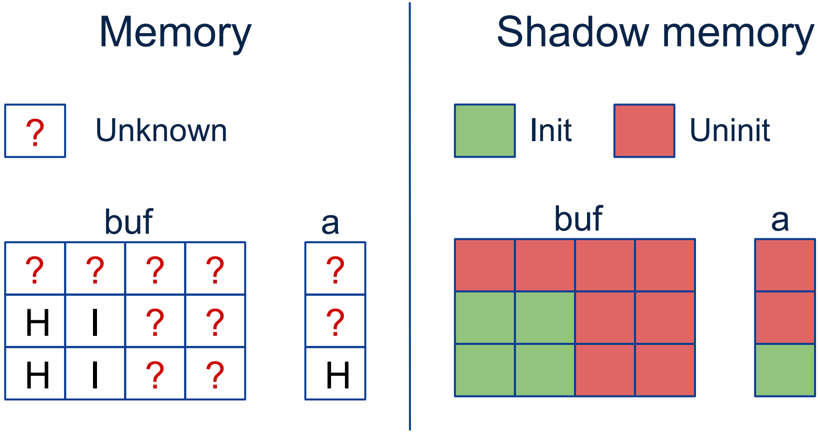
🟩 Init 🟥 Uninit

buf				a
🟥	🟥	🟥	🟥	🟥
🟩	🟩	🟥	🟥	🟥
🟩	🟩	🟥	🟥	🟩

Check *a*’s shadow

MSan - Workflow

```
void foo(){
    char buf[4], a;
    read(0, buf, 4);
    a = buf[0];
    if(a==MAGIC_BYTE)
        puts("Hello world!");
}
```



Memory

buf				a
?	?	?	?	?
H	I	?	?	?
H	I	?	?	H

Shadow memory

buf				a
🟥	🟥	🟥	🟥	🟥
🟩	🟩	🟥	🟥	🟥
🟩	🟩	🟥	🟥	🟩

Check *a*’s shadow

Total: **5 operations**

Binary Detection

- Detect UUM errors at the binary level
  - Similar workflow as MSan
  - Much more instrumentation

Binary Detection

- Detect UUM errors at the binary level
  - Similar workflow as MSan
  - Much more instrumentation

Pros:

- More generic
  - No recompilation
  - Closed-source software

Binary Detection

- Detect UUM errors at the binary level
  - Similar workflow as MSan
  - Much more instrumentation

Pros:

- More generic
  - No recompilation
  - Closed-source software

Cons:

- Slow (10-20x slowdown)
  - Shadow propagation is **much** harder
- No fuzzing compatibility

QMSan - overview

- Binary-based multi-layered solution to detect UUM errors
  - based on the QEMU emulator
  - fuzzing-compatible

QMSan - overview

- Binary-based multi-layered solution to detect UUM errors
  - based on the QEMU emulator
  - fuzzing-compatible

- Three main components:

QMSan - overview

- Binary-based multi-layered solution to detect UUM errors
  - based on the QEMU emulator
  - fuzzing-compatible

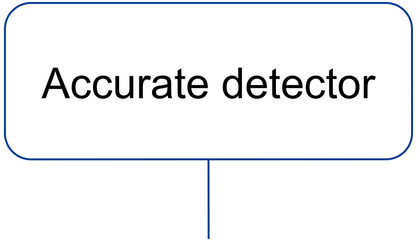
- Three main components:

Accurate detector

QMSan - overview

- Binary-based multi-layered solution to detect UUM errors
  - based on the QEMU emulator
  - fuzzing-compatible

- Three main components:



Accurate detector

Similar to binary UUM detectors

Very Accurate, but very slow

QMSan - overview

- Binary-based multi-layered solution to detect UUM errors
  - based on the QEMU emulator
  - fuzzing-compatible

- Three main components:

Accurate detector

Run-time module

QMSan - overview

- Binary-based multi-layered solution to detect UUM errors
  - based on the QEMU emulator
  - fuzzing-compatible

- Three main components:

Accurate detector

Run-time module

Supports UUM detection with shadow memory management

QMSan - overview

- Binary-based multi-layered solution to detect UUM errors
  - based on the QEMU emulator
  - fuzzing-compatible

- Three main components:

Accurate detector

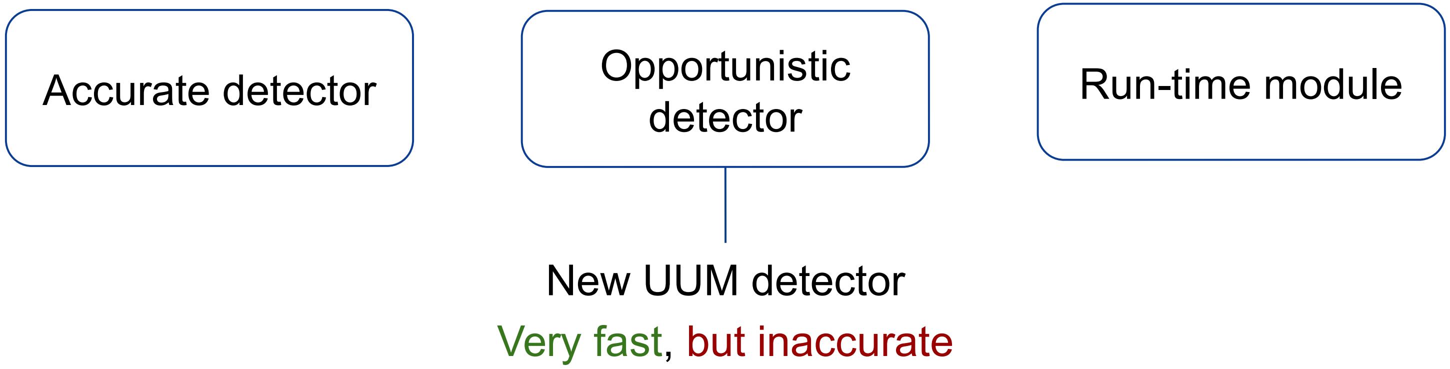
Opportunistic detector

Run-time module

QMSan - overview

- Binary-based multi-layered solution to detect UUM errors
  - based on the QEMU emulator
  - fuzzing-compatible

- Three main components:



Accurate detector

Opportunistic detector

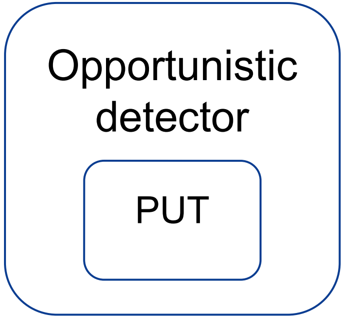
Run-time module

New UUM detector

Very fast, but inaccurate

QMSan - Workflow

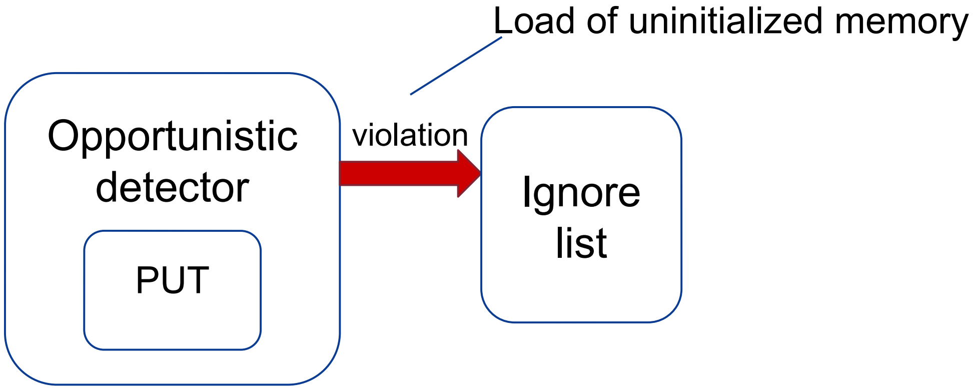
QMSan - Workflow



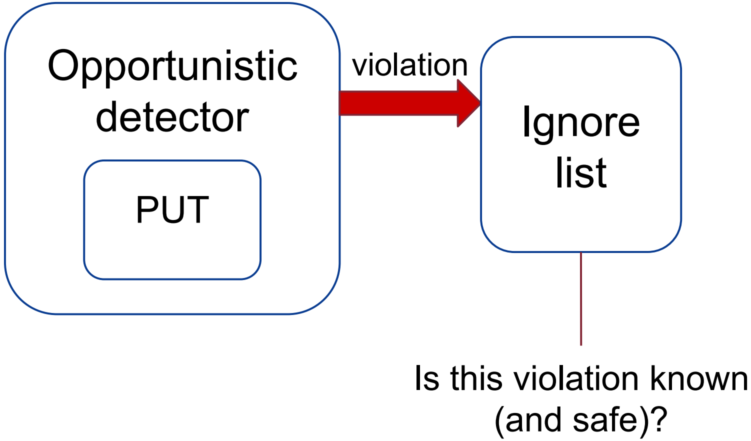
Opportunistic detector

PUT

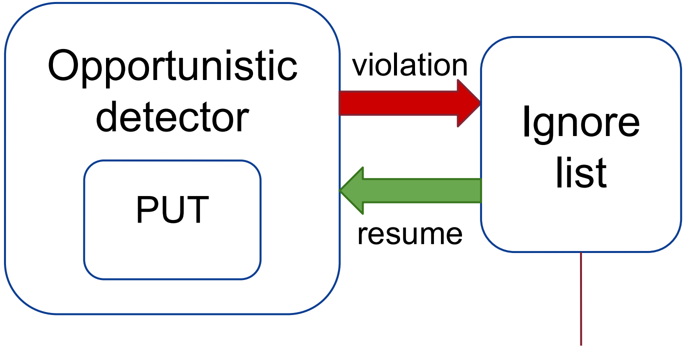
QMSan - Workflow



QMSan - Workflow



QMSan - Workflow



Opportunistic detector

PUT

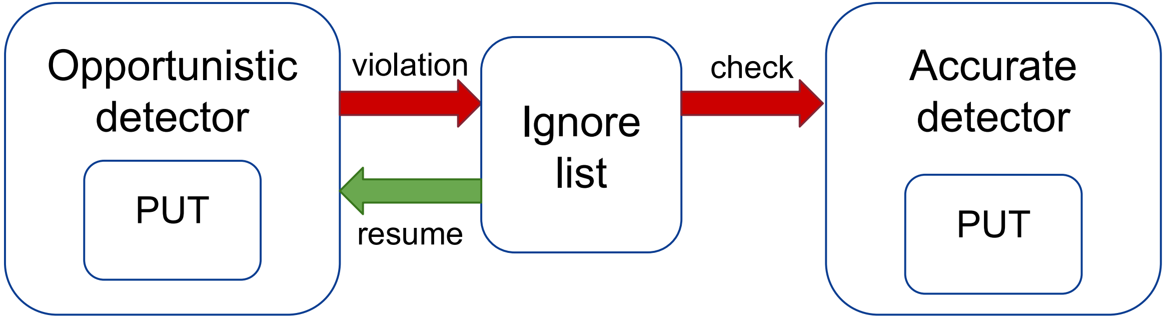
violation

resume

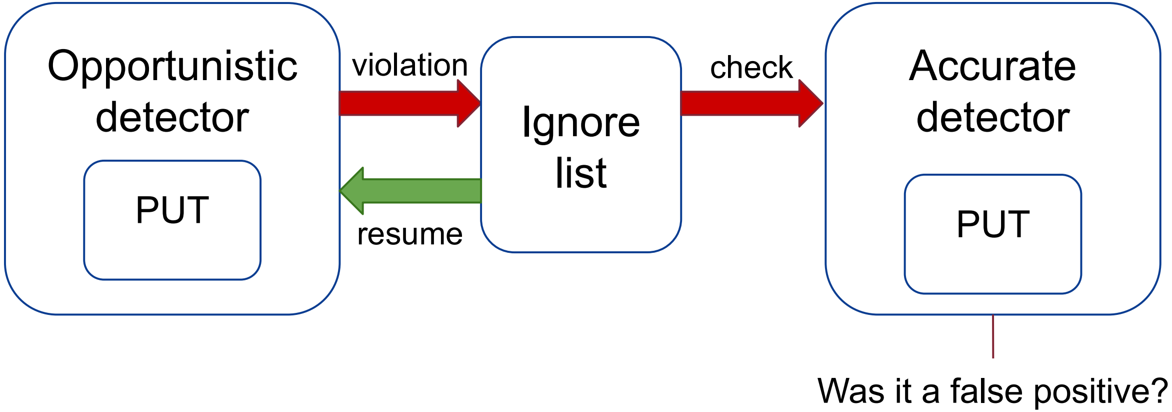
Ignore list

Is this violation known (and safe)?

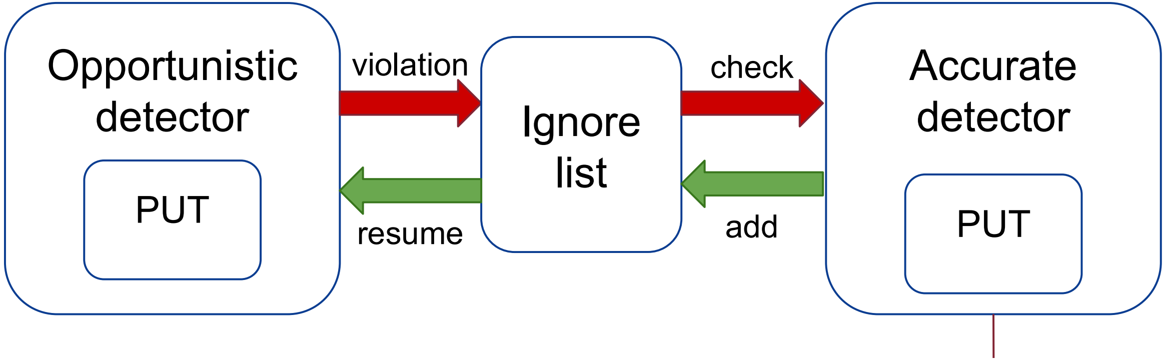
QMSan - Workflow



QMSan - Workflow



QMSan - Workflow



Opportunistic detector

PUT

violation

resume

Ignore list

check

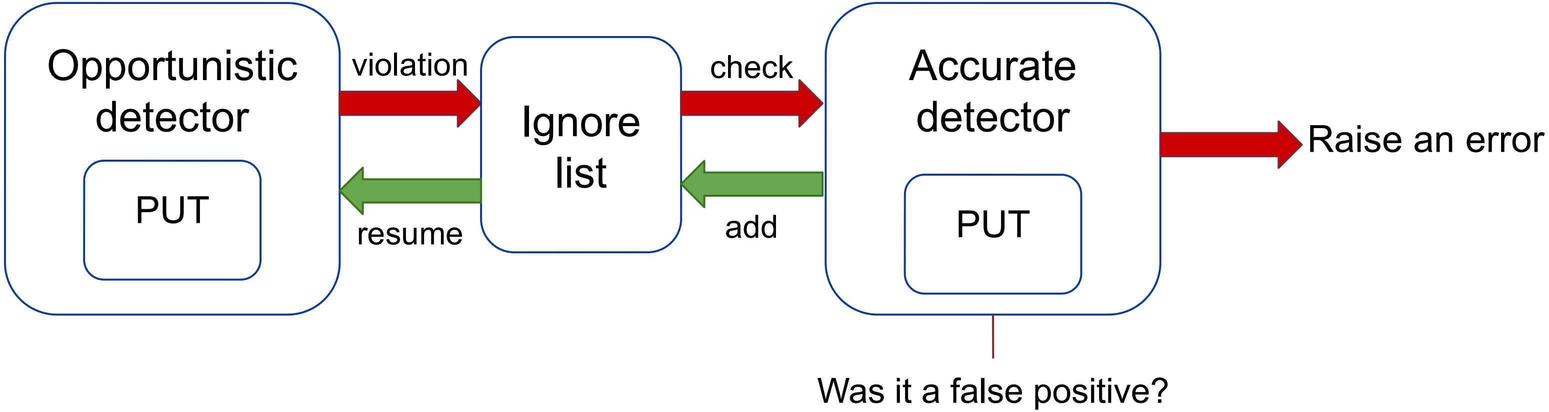
add

Accurate detector

PUT

Was it a false positive?

QMSan - Workflow



Opportunistic detector

PUT

violation

resume

Ignore list

check

add

Accurate detector

PUT

Raise an error

Was it a false positive?

QMSan - Opportunistic Detector

Key intuition: Most loads of uninitialized memory are safe…

We don’t need to check them every time!

QMSan - Opportunistic Detector

Key intuition: Most loads of uninitialized memory are safe…

We don’t need to check them every time!

Opportunistic detection:

QMSan - Opportunistic Detector

Key intuition: Most loads of uninitialized memory are safe…

We don’t need to check them every time!

Opportunistic detection:

- Only check memory accesses (R/W)

QMSan - Opportunistic Detector

Key intuition: Most loads of uninitialized memory are safe…

We don’t need to check them every time!

Opportunistic detection:

- Only check memory accesses (R/W)

Write: initialize shadow

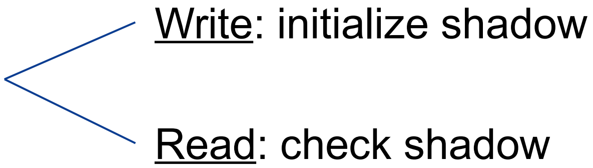
QMSan - Opportunistic Detector

Key intuition: Most loads of uninitialized memory are safe…

We don’t need to check them every time!

Opportunistic detection:

- Only check memory accesses (R/W)



Write: initialize shadow

Read: check shadow

QMSan - Opportunistic Detector

Key intuition: Most loads of uninitialized memory are safe…

We don't need to check them every time!

Opportunistic detection:

- Only check memory accesses (R/W)
- When a **violation** occurs:

Write: initialize shadow

Read: check shadow

QMSan - Opportunistic Detector

Key intuition: Most loads of uninitialized memory are safe…

We don’t need to check them every time!

Opportunistic detection:

- Only check memory accesses (R/W)
  - Write: initialize shadow
  - Read: check shadow
- When a **violation** occurs:

Known: keep executing

QMSan - Opportunistic Detector

Key intuition: Most loads of uninitialized memory are safe…
We don’t need to check them every time!

Opportunistic detection:

- Only check memory accesses (R/W)
- When a **violation** occurs:

Write: initialize shadow

Read: check shadow

Known: keep executing

Not Known: Use propagation to check and **remember for next time**

QMSan - Ignore list

- Used to remember violations
  - Good policies are needed

QMSan - Ignore list

- Used to remember violations
  - Good policies are needed

- Violations Are added considering:

QMSan - Ignore list

- Used to remember violations
  - Good policies are needed

- Violations Are added considering:

Instruction’s address

QMSan - Ignore list

- Used to remember violations
  - Good policies are needed

- Violations Are added considering:

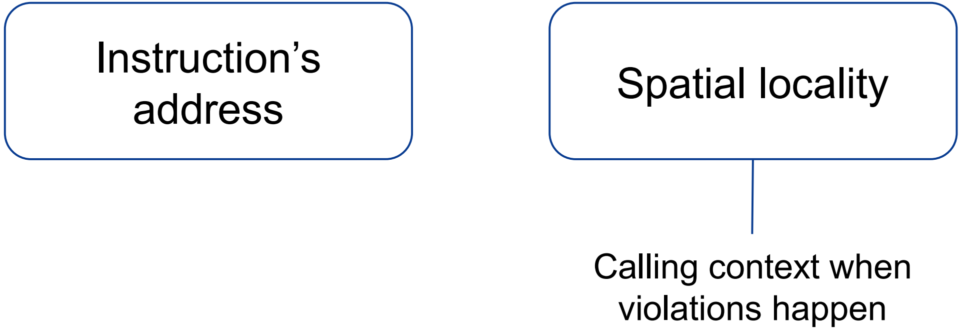
Instruction's address

Spatial locality

QMSan - Ignore list

- Used to remember violations
    - Good policies are needed

- Violations Are added considering:



Instruction's address

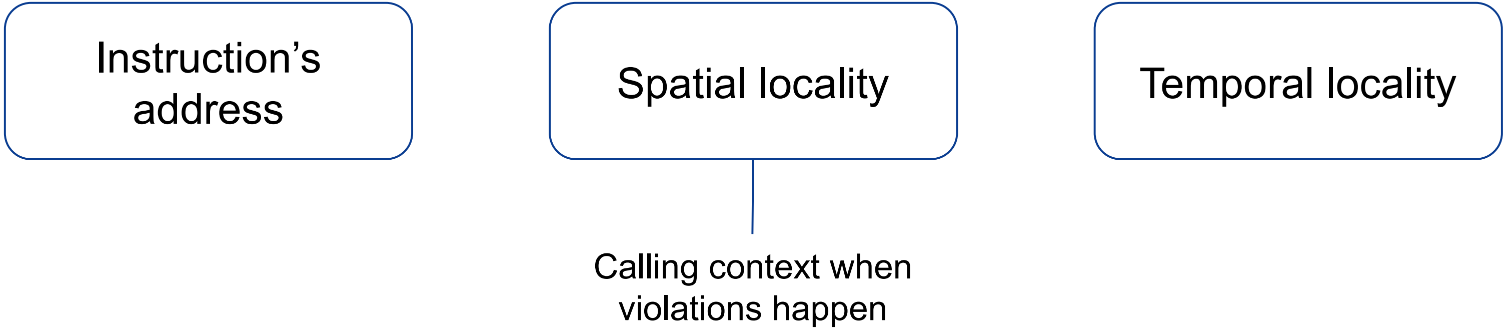
Spatial locality

Calling context when violations happen

QMSan - Ignore list

- Used to remember violations
    - Good policies are needed

- Violations Are added considering:



Instruction's address

Spatial locality

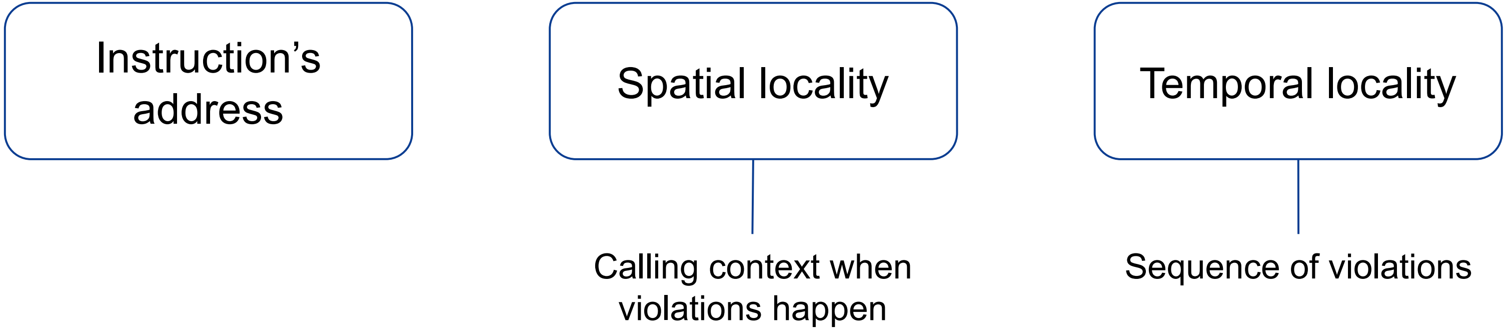
Temporal locality

Calling context when violations happen

QMSan - Ignore list

- Used to remember violations
  - Good policies are needed

- Violations Are added considering:



Instruction’s address

Spatial locality

Calling context when violations happen

Temporal locality

Sequence of violations

Evaluation - Bugs

Dataset:

- 9 closed-source binaries
  - 5 projects, multiple versions
- 10 open-source programs (from OSS-Fuzz)

Methodology:

- 72 hours runs
- 3 runs

Evaluation - Bugs

Dataset:

- 9 closed-source binaries
  - 5 projects, multiple versions
- 10 open-source programs (from OSS-Fuzz)

Subject	Vendor	Version	Bugs
cuobjdump	NVIDIA	12.3	2
cuobjdump	NVIDIA	12.4	0
nconvert	XnView Software	7.136	5
nconvert	XnView Software	7.155	4
nvdisasm	NVIDIA	12.3	7
nvdisasm	NVIDIA	12.4	3
pngout	Ken Silverman	Jan 15 2020	2
rar	rarlab	6.11	1
rar	rarlab	7.0	3
Total			27

Methodology:

- 72 hours runs
- 3 runs

Evaluation - Bugs

Dataset:

- 9 closed-source binaries
  - 5 projects, multiple versions
- 10 open-source programs (from OSS-Fuzz)

Subject	Vendor	Version	Bugs
cuobjdump	NVIDIA	12.3	2
cuobjdump	NVIDIA	12.4	0
nconvert	XnView Software	7.136	5
nconvert	XnView Software	7.155	4
nvdisasm	NVIDIA	12.3	7
nvdisasm	NVIDIA	12.4	3
pngout	Ken Silverman	Jan 15 2020	2
rar	rarlab	6.11	1
rar	rarlab	7.0	3
Total			27

Methodology:

- 72 hours runs
- 3 runs

Subject	Version	Bugs
libredwg	763d702	3
gpac	205bfe3	1
assimp	b71b8f7	2
libdwarf	6178ba8	2
serenity	7914383	1
opensc	fe2c1c8	5
ntopng	8786f06	1
upx	3495d1a	2
radare2	cfe5806	0
libucl	5c58d0d	0
Total		17

Evaluation - Performance

Dataset:

- 8 common fuzzing benchmarks (from Google’s FTS)

Methodology:

- 72 hours runs
- 3 runs

Evaluation - Performance

Dataset:

- 8 common fuzzing benchmarks (from Google’s FTS)

Methodology:

- 72 hours runs
- 3 runs

Project	QMSan		
Name	vs AFL-cc	vs MSan	vs QEMU
c-ares	2,20	1,05	1,04
guetzli	3,17	1,24	1,41
json	2,69	1,24	1,12
libxml2	3,41	0,90	1,42
openssl	19,84	8,24	4,68
pcre2	3,18	1,42	1,40
re2	3,35	1,48	1,48
woff2	2,86	1,34	1,20
geomean	3,75	1,55	1,51

Conclusions

Conclusions

- Detecting UUM errors is a challenging task due to the **slowdown introduced by shadow propagation**

Finanziato dall'Unione europea NextGenerationEU

Ministero dell'Università e della Ricerca

Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA

SERICS SECURITY AND RIGHTS IN THE CYBERSPACE

Conclusions

- Detecting UUM errors is a challenging task due to the **slowdown introduced by shadow propagation**

- We presented a new design that **drastically limits shadow propagation** at the binary level.
  - 44 new bugs (4 CVEs)
  - 1.51x slowdown over QEMU

Conclusions

- Detecting UUM errors is a challenging task due to the **slowdown introduced by shadow propagation**
- We presented a new design that **drastically limits shadow propagation** at the binary level.
  - 44 new bugs (4 CVEs)
  - 1.51x slowdown over QEMU

https://github.com/Heinzeen/qmsan