04834580 Software Engineering (Honor Track) 2025-26

Dynamic Analysis and Fuzzing

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

Definition (Dynamic Analysis)

Analysis of a program performed by **executing** it (possibly with instrumentation), and observing its runtime behaviour on concrete inputs.

Why Dynamic Analysis?

Many bugs are hard to detect by compilers or human reviewers:

- Out-of-bounds reads and writes
- Use-after-free, double-free
- Memory leaks
- Use of uninitialised memory
- Data races between threads
- Information flow from untrusted inputs to sensitive sinks

The Sanitizer Family

Compiler-supported, lightweight dynamic checkers built into Clang and GCC:

- **AddressSanitizer (ASan)** — spatial and temporal memory errors
- **LeakSanitizer (LSan)** — memory leaks at process exit
- **MemorySanitizer (MSan)** — reads of uninitialised memory
- **ThreadSanitizer (TSan)** — data races
- **UndefinedBehaviorSanitizer (UBSan)** — C/C++ undefined behaviour

Activated with a single compiler flag, e.g.

```
clang++ -fsanitize=address -g program.cpp
```

AddressSanitizer: Motivating Example

北京大学

A heap buffer overflow is silent without instrumentation:

```
int *array = new int[100];
array[0] = 0;
int res = array[argc + 99];    // out-of-bounds read
```

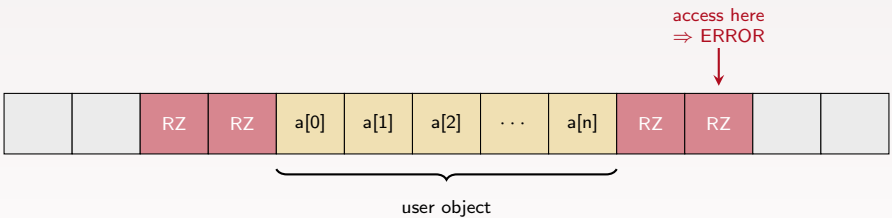
The address `array[100]` usually still belongs to the process, so the read *seems* to succeed and returns garbage. The bug may manifest much later as a wrong result or a crash in unrelated code.

AddressSanitizer [1] catches it at the exact instruction that performs the bad access.

Key Idea: Red Zones around Allocations

北京大学

Every heap (and stack) object is surrounded by **red zones** — small ranges of memory marked as "poisoned". Any access that touches a poisoned byte is reported as an error.



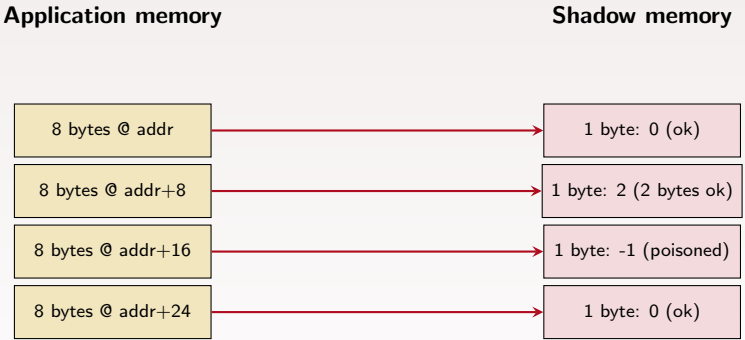
Freed memory is also poisoned and held in *quarantine* for a while so that use-after-free is detected even when the slot has not yet been recycled.

Definition (Shadow Memory)

A second region of memory in which each byte (or block of bytes) of the program's memory has a corresponding *shadow* byte describing its state (addressable, poisoned, partially initialised, ...).

AddressSanitizer maps every 8-byte aligned chunk of application memory to a single shadow byte, encoding how many of its 8 bytes are addressable:

`0`	all 8 bytes addressable
`1..7`	only the first k bytes addressable
negative	poisoned (red zone, freed, stack-after-return, ...)



address $\rightarrow$ shadow: $S(a) = (a \gg 3) + \text{Offset}$

What the Compiler Inserts

北京大学

Conceptually, every load/store is rewritten:

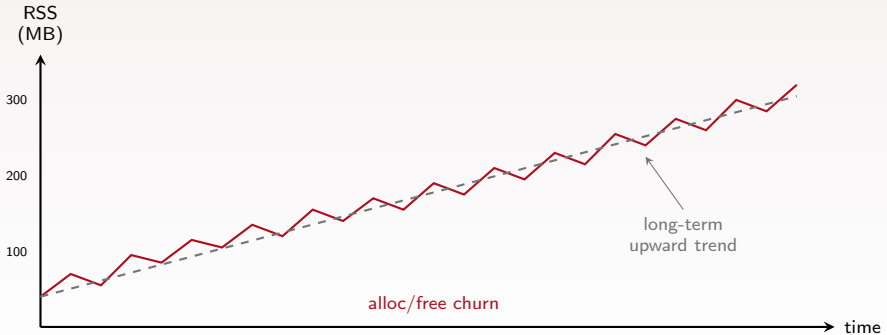
```
// original
*addr = value;

// instrumented
ShadowAddr = (addr >> 3) + Offset;
if (*ShadowAddr != 0 && SlowCheck(addr, *ShadowAddr))
    ReportError(addr);
*addr = value;
```

- One shift, one load, one compare on the fast path.
- Typical slowdown: $\approx 2\times$; memory overhead ≈ 1.5–$3\times$.
- Detects: heap/stack/global overflows, use-after-free, use-after-return, double-free, invalid `free`.

Definition (Memory Leak)

A memory leak occurs when a program allocates memory on the heap but loses every reference to it before deallocating it. The memory remains allocated for the lifetime of the process, yet is no longer reachable by the program.



```c
void *p;
int main() {
    p = malloc(7);
    p = 0;          // last pointer to the buffer is overwritten
    return 0;
}
```

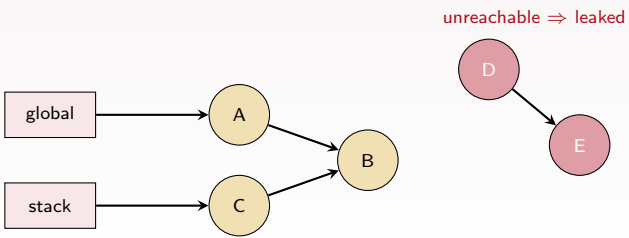
After the assignment, no live variable holds the address returned by `malloc`. The 7 bytes are *lost*: still allocated, but unreachable. LeakSanitizer reports it at process exit.

How Leak Detection Works

北京大学

Leak detection is a *mark-and-sweep* pass over the heap performed at program exit (or on demand):

1. **Roots**: registers, thread stacks, global variables, thread-local storage — anything that can hold a live pointer.
2. **Mark**: scan the roots word-by-word; any word whose value falls inside a known live allocation marks that allocation as *reachable*, then recursively scan it.
3. **Sweep**: every allocation that was never marked is a leak; report its size and the stack trace of its allocation.



MemorySanitizer: Uninitialised Reads

北京大学

C and C++ leave `malloc`'d and stack-allocated memory *undefined*. Reading it before writing it is undefined behaviour, but very hard to spot:

```
int *p = (int*)malloc(sizeof(int));
if (*p == 42) {             // reads uninitialised value
    launch_missiles();
}
```

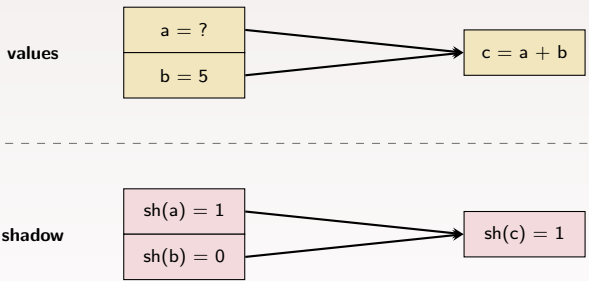
MemorySanitizer [2] tracks the *definedness* of every bit of memory and every register and reports the first *branch* or *syscall* that depends on uninitialised data.

Bit-level Shadow Memory

北京大学

MSan keeps a shadow *bit* for every program bit:

- shadow bit $= 0 \Rightarrow$ the corresponding bit has been initialised by a store;
- shadow bit $= 1 \Rightarrow$ the corresponding bit is poisoned (uninitialised).



shadow propagates in parallel: $\mathrm{sh}(a + b) = \mathrm{sh}(a) \vee \mathrm{sh}(b)$

An error is reported only when poisoned data **influences observable behaviour** — a conditional jump, a pointer dereference, or a system call argument.

```
int v = 0;
void foo() { v++; }
void bar() { v++; }

int main() {
    std::thread t1(foo), t2(bar);
    t1.join(); t2.join();
    std::cout << v;        // 1 or 2 ?
}
```

Two threads write to the same location without synchronisation. The program may print 1 or 2, depending on the thread schedule. ThreadSanitizer [3] detects such races even when they do not actually misbehave on this particular run.

The Happens-Before Relation

Given events A and B, we say A **happens-before** B ($A \to B$) iff at least one of the following holds:

- **Program order:** A and B are executed by the same thread t, and t executes A before B.
- **Inter-thread message:** A and B are executed by different threads, and A is a *send* event paired with a *receive* event B.
- **Transitivity:** there exists an event C such that $A \to C$ and $C \to B$.

A send may have a single receiver, multiple receivers, or none.

- **Send events:** `unlock(m)`, `notify()`, `notifyAll()`, `thread.start()`, atomic store-release.
- **Receive events:** `lock(m)`, `wait()`, `join()`, atomic load-acquire.

Definition (Data Race)

A data race occurs when two events A and B satisfy all three:

- A and B are **concurrent**: neither $A \to B$ nor $B \to A$;
- A and B access the **same memory word**;
- at least one of the two accesses is a **write**.

ThreadSanitizer instruments every memory access and every send/receive event. For each memory word it remembers recent accesses; on each new access it checks the three conditions above and reports a race if all hold.

Definition (Dynamic Taint Analysis [4])

Track, during execution, whether each value in memory or in a register is **derived from a designated source** ("tainted"), and report whenever a tainted value reaches a designated sink.

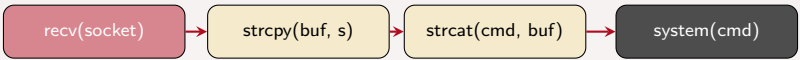
Three ingredients:

- **Sources** — where taint enters: `read`, `recv`, command-line arguments, HTTP request bodies, environment variables.
- **Propagation rules** — how operations transmit taint from inputs to outputs.
- **Sinks** — where tainted data must not flow: `system`, `exec`, SQL execution, indirect jumps.

Each byte of memory and each register has a one-bit shadow "taint" tag. The rules for the tag are intuitive:

`c = a + b`	$T(c) = T(a) \vee T(b)$
`c = a & 0`	$T(c) = 0$ (constant)
`*p = a`	$T(mem[p]) = T(a)$
`a = *p`	$T(a) = T(mem[p])$
`a = read(...)`	$T(a) = 1$ (source)
`system(s)`	if $T(s) = 1$ then *alert*

Taint Flow Example



tainted source

sensitive sink

End-to-end flow tainted user input → `system()`: command-injection alert

Implicit Flows

北京大学

Taint analysis as described tracks only *explicit* data flow (assignments, arithmetic). It can miss **implicit** flows:

```
if (tainted == 0) y = 0; else y = 1;    // y depends on tainted
```

Tracking implicit flows requires reasoning about control dependence and is much more expensive — and tends to over-taint everything.

Where Taint Analysis Is Used

北京大学

- **Vulnerability detection**: SQL injection, command injection, XSS, format-string bugs — any "input reaches dangerous API" pattern [5].
- **Privacy auditing**: track sensitive data (passwords, GPS, contacts) on Android (TaintDroid) and stop it from leaving the device.
- **Fuzzing guidance**: which input bytes influence which branches? Used by hybrid fuzzers (VUzzer, Angora) to mutate only the bytes that matter.
- **Reverse engineering**: figure out which fields of a packet drive a parser's behaviour.

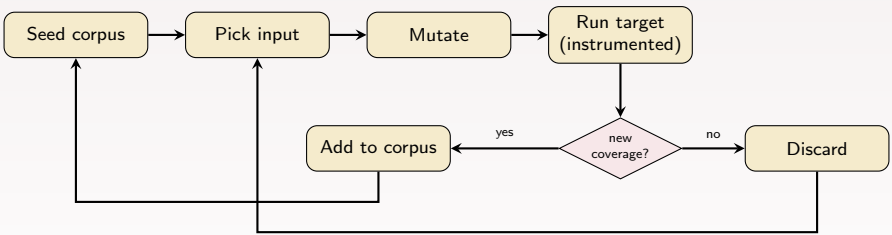
Definition (Fuzzing)

A testing technique in which a program is repeatedly executed on **automatically generated, often malformed, inputs**, looking for crashes, hangs, or sanitizer reports.

Origin: Miller et al. in 1990 [6] fed random bytes to UNIX command-line utilities and managed to crash 25–33% of them. Modern fuzzers are far more targeted.

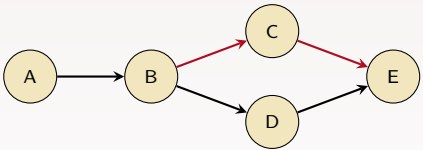
- **Black-box**: treat the program as a closed box, just throw inputs at it.
- **White-box**: use program analysis (e.g. symbolic execution [7]) to derive new inputs.
- **Grey-box**: cheap instrumentation collects *some* runtime feedback (typically code coverage) and uses it to steer mutation. **AFL is the canonical example.**

Idea behind AFL [8]: an input that triggers *new* code coverage is interesting — save it and mutate it. Inputs that retread known territory are discarded.

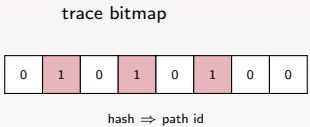


Ideally a fuzzer would track every distinct **control-flow path** through the program. The number of paths is exponential in the number of branches, so this is infeasible.

AFL uses a much cheaper proxy: a fixed-size **trace bitmap** of edge-hit counts. The hash of this bitmap is the input's **path id**.



execution: A → B → C → E



Two inputs with the same bitmap are deemed to take the "same path".

How AFL Measures Coverage

At compile time AFL inserts a tiny snippet at every basic-block edge:

- A 64 KB **shared bitmap** is updated by every executed edge. Counts are bucketed (1, 2, 3, 4–7, 8–15, ...) so that "hit a loop 100 times vs. 101" is not a new edge but "1 vs. 8" is.
- After execution the fuzzer compares the bitmap against the global one. Any new edge or new bucket ⇒ keep the input.

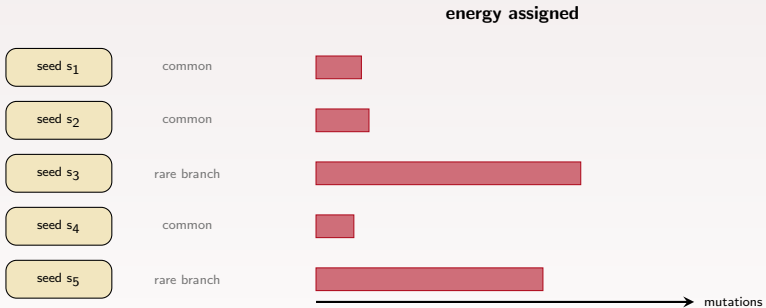
Mutation Strategies

北京大学

AFL combines several mutation strategies:

- **Bit/byte flips** at sliding offsets.
- **Arithmetic** on integers ($\pm 1, \pm 2, \ldots, \pm 35$).
- **Interesting values** (0, -1, `INT_MIN`, `INT_MAX`, ...).
- **Dictionary** tokens (e.g. HTTP keywords for a web parser).
- **Splice / havoc**: combine two corpus inputs and apply random stacked mutations.

A **power schedule** [9] assigns an *energy* (number of mutations) to each seed.

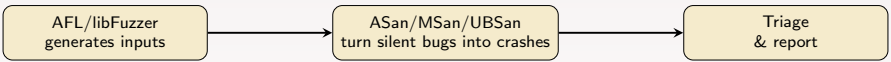


- Seeds that hit *rare* edges get more mutations.
- Seeds revisiting popular regions are starved.
- Result: faster discovery of new coverage on the same budget.

Putting It Together: Find Bugs Faster

北京大学

Modern fuzzing campaigns combine the techniques from this lecture:



Summary

- **Dynamic analysis** catches bugs by observing real executions; the price is non-trivial runtime overhead.
- **Sanitizers** use *shadow memory* to give every byte a tag (poisoned, defined, tainted, ...) and check it on every access.
  - ASan: red zones ⇒ spatial/temporal memory errors.
  - LSan: mark-and-sweep at exit ⇒ leaks.
  - MSan: bit-shadow ⇒ uninitialised reads.
  - TSan: vector clocks ⇒ data races.
- **Dynamic taint analysis** follows "where does this input go?" from sources to sensitive sinks.
- **Coverage-based fuzzing** (AFL) uses cheap edge instrumentation as feedback to evolve a corpus that covers ever more of the program.
- Sanitizers and fuzzing are most powerful **together**.

References I

[1] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov.
AddressSanitizer: A fast address sanity checker.
In *USENIX Annual Technical Conference (ATC)*, pages 309–318, 2012.

[2] Evgeniy Stepanov and Konstantin Serebryany.
MemorySanitizer: Fast detector of uninitialized memory use in C++.
In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, 2015.

[3] Konstantin Serebryany and Timur Iskhodzhanov.
ThreadSanitizer: Data race detection in practice.
In *Workshop on Binary Instrumentation and Applications (WBIA)*, pages 62–71, 2009.

References II

[4] Edward J Schwartz, Thanassis Avgerinos, and David Brumley.
All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask).
In *IEEE Symposium on Security and Privacy (S&P)*, pages 317–331, 2010.

[5] James Newsome and Dawn Song.
Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.
In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[6] Barton P Miller, Louis Fredriksen, and Bryan So.
An empirical study of the reliability of UNIX utilities.
Communications of the ACM, 33(12):32–44, 1990.

[7] Patrice Godefroid, Michael Y Levin, and David Molnar.
Automated whitebox fuzz testing.
In *Network and Distributed System Security Symposium (NDSS)*, 2008.

References III

[8] Michał Zalewski.
American Fuzzy Lop (AFL).
https://lcamtuf.coredump.cx/afl/, 2014.

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury.
Coverage-based greybox fuzzing as Markov chain.
In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1032–1043, 2016.