Intro

Manual testing: programmers construct test cases, either to achieve high coverage, or in response to known bugs
- **High effort**: time consuming to writing high coverage tests
- **Limitations of human thought**: Human testers do not tend to spot intricate, unusual input combinations that may lead to failure

General bugs: Integer overflow, Division or modulo by zero, Memory bugs, Out-of-bounds array access, Null pointer dereference, Double free, Access after free
Concurrency bugs: Data race, Deadlock
Termination bugs: Infinite loop, Unbounded recursion
Functional bugs: Incorrect algorithm
Undefined behavior: result of executing code whose behavior is not prescribed by the language standard, e.g. Buffer overflows, Null-pointer dereferences, Division/modulo by zero, Uninitialized reads, Shifting by a negative number
General properties: know what to check
- often involve quantifiers, present a challenge to provers

Functional properties: don't know a priori what to check
- requires specifications

Safety properties: something bad does not happen
Liveness properties: something good happens eventually
Dynamic analysis: Involves running programs (either directly or through emulation) and collecting information about executions
- **Pros**: precise, scalable (proportional to regular execution)
- **Cons**: Requires whole system (need a test driver), environment or simulator, quality of test inputs important
- e.g. Software fuzzing, Compiler sanitizers, Dynamic symbolic execution, Mutation testing

Static Analysis: Reasoning about program executions without actually running the program
- **Pros**: find bugs not found by existing test cases, high cov., applicable to incomplete systems, scalable if modular
- **Cons**: either price & expensive or fast & imprecise
- e.g. compilers, Open-source, commercial, company tools

False positive: warns about a problem that can't actually occur
- Regarded as the main problem

Imprecise: reports lots of false positives
Incomplete: may report false positives
Complete tool: never reports a false positive
False negative: reports absence of problems when actually problems can occur
Unsound: may report false negatives
Sound tool: never reports a false negative

Fuzzing

Fuzzing: Testing a system (SUT) using inputs that are wholly or partly randomly-generated to find inputs that make the SUT behave in interesting ways
- Needs way of getting inputs & oracle decides interesting
- e.g. crash? New code? Error? Assertion fail? Correct?
- Usually an oracle for correctness does not exist

Generation-based fuzzer: produces inputs from nothing
Mutation-based fuzzer: produces an input through modification and/or combination of existing inputs
Randomized input: totally invalid < Somewhat malformed (tokens) < with a degree of validity (mal-typed) < High integrity
Dumb fuzzer: generates or mutates inputs without knowledge of the SUT's input domain
Smart fuzzer: uses some information about the expected input format, e.g. a grammar, model or protocol description
Compiler sanitizers: Compiler-based dynamic analyses that can reliably detect various kinds of errors
Grammar-based Fuzzing: start symbol Expr -> rand. pick a non-terminal symbol -> rand. pick a production rule
- Must consider probability distributions

Feedback-directed fuzzing: inform a fuzzer's input generation or mutation procedure by observing how the SUT behaves
- input can be flagged if it makes SUT do things noteworthy
  - e.g. coverage, memory↑, log unseen msg, syscall
- **Generation-based**: try to generate with similar properties
- **Mutation-based**: prioritize input as candidate for new

Black-box Fuzzing: SUT is executed in unmodified form
Pros: applicable to closed-source SUT, SUT runs at full speed
Cons: Feedback can only observe externally-visible SUT behavior
Grey-box Fuzzing: Instrumentation applied to SUT, to yield information about internal workings, typically coverage
Pros: feedback fuzzer has more information to work with
Cons: compile-time instrumentation requires SUT source
White-box Fuzzing: Program analysis via constraint solving used to generate inputs that exercise provably different parts of SUT
- Symbolic/concolic execution

Pros: reach hard-to-hit parts of the SUT
Cons: limited scalability b/c of overhead of solving, heavyweight
AFL: mutation-based, dumb, grey-box fuzzer
Load initial test cases to queue -> Take from queue -> trim test case to smallest w/o alter behavior -> add mutant to queue if causes new state transition -> take another from queue -> …
Mutation Strategies: Walking bit-flips & byte-flips, ints++/ints--, insert known ints, deleting or mem-setting parts of input file, splicing (concatenating)
Constraint solver: a tool that can determine the satisfiability of certain types of formulas
SAT solvers: Boolean formulas (typically given in CNF)
SMT solvers: formulas in a given theory, e.g., int, BVs, arrays
Worth fixing: any crash in security & safety-critical systems, frequently-occurring bugs
Not be fixed: limited budget, unimportant, fix lead to errors

Compiler Fuzzing and Derived Test Oracles

Compilers: part of our trusted development base; bugs can affect quality, reliability & security of current & future programs
Types of bugs: Crash bug, Miscompilation
Why compilers hard to test: Undefined, unspecified and implementation-defined behavior, no oracle, Nondeterminism, lack of clear language semantics
Undefined behavior: the standard imposes no requirement
Unspecified behavior: the standard defines a set of allowable behaviors, e.g. order of evaluation of function arguments
Implementation-defined behavior: the standard defines a set of allowable behaviors and compiler must choose and document
- e.g. propagation of high-order bit when a signed (negative) integer is shifted right

There is no oracle for compilers: allows optimizations producing slightly different results; writing the golden reference is hard
Non-testable Program: There does not exist an oracle, or it is theoretically possible, but practically too difficult
Derived Oracle - Crosscheck Implementations: Compile a program with many compilers & Compare results
- Req: deterministic, no UB, multiple compilers, agree on IB

Derived oracle - Crosscheck Equivalent Programs: Compile equivalent programs with one compiler & Compare results
- Req: equivalent programs - Vanilla & optimized versions or programs derived from one another via transformations, deterministic, no UB

Random program: crash-proof, correct rejection & execution
- Desirably a range of language features, interestingly combined, terminate, observable output, non-trivial

Create Program from Nothing: Generates an abstract syntax tree via a random walk, Ensures properties, e.g. types, no UB
- Challenges: large sizes but not exceeding resource limits, mix of features, termination (eventual but not premature)

Csmith: generates C programs free from UB, suitable for differential testing; print a checksum on termination
- smart, generation, black-box (no feedback), differential

Avoiding Undefined Behavior: safe math macros
- e.g. ((e2) == 0 ? (e1) : (e1) / (e2)); simple but idiomatic
- Memory safety ensured by a more involved effect analysis

About 10% of programs generated by Csmith nonterminating
- easy to deal with using timeouts

Crash testing: very useful; Finding miscompilations: less so
- cannot be debugged via huge programs

Equivalence Modulo Inputs (EMI) Testing: execute I with D (not covered by I) altered on P1 & P2
- Mutation, smart, black-box (no feedback), metamorphic

Metamorphic testing: test R(SUT(x), SUT(f(x)) to hold
- f is a metamorphic transformation, e.g. x → x + 2π for sin

Metamorphic Compiler Testing: returns output generated by P when compiled and run on input I
- For EMI, f(P) changes I-dead code

Test Case Reduction: reduce while preserve interesting program
while worthPersevering(history, P):
- Q = applyReduction(history, P)
- interesting ? P = Q && history.add() : history.add()

applyReduction: Remove lines, replace a complex expression with a simpler one, inlining, remove an un-invoked function
interesting: Compiler fatal error, timeout, Wrong code bugs
C-Reduce: Line- and token-based transformations

Undefined Behavior

e.g. Saturating Add, Guarded Abs (for -2^{31}, on x86 = -2^{31}), null
UB is global; compiler optimizations may cause code to behave unexpectedly under UB; Compiler assumes no UB
UB to cater diverse hardware:
- for signed addition overflow, x86 wraps, MIPS traps
- 1<<32 on x86 = 1, on ARM & PowerPC = 0
- dereferencing 0 on x86 causes runtime exception, on ARM holds exception handlers
- Standardizing is expensive for some platforms

UB to allow powerful optimizations:
- e.g. change type of i to long to avoid extending only works if i++ doesn't overflow

UB examples: reading from uninitialized variables, pointing >1 past end of an array even w/o dereferencing, aliasing between pointers with different types, memcpy with overlap buffers
Impact of UB on program analysis: many program analysis tools re-use compiler front-ends → compiler front-end can generate arbitrary code for UB → relying on compiler front-ends that exploit UBs can render a would-be sound tool unsound

Unstable Code

Optimization-unstable code: code that is unexpectedly discarded by compiler optimizations due to UB in the program
- any fragment that is reachable only by inputs that trigger undefined behavior is unstable code

Stack: a static checker to identifies unstable code
Evolution of optimizations: more unstable code discarded as the compilers evolve to adopt new optimizations, some even discard unstable code at the lowest level of optimization -O0
C*: denote a C dialect that assigns well-defined semantics to code fragments that have undefined behavior in C
- P⇝P[e/e'] is legal if P[e/e'] has same result or P has UB
- code fragment e in program P is unstable iff there exists e' such that P ⇝ P[e/e'] is legal under C but not under C*

Stack: 1. run O w/o taking advantage of UB; 2. run O again taking advantage of UB; if O optimizes extra code in the second phase, code is unstable
- Need to be sufficiently aggressive to not miss

R_e(x): denote e's reachability condition, which is true iff e will execute under input x
U_e(x): denote e's undefined behavior condition, which indicates whether e exhibits undefined behavior on input x, assuming C semantics
Well-defined program assumption:
- $\Delta(x) = \bigwedge_{e \in P} R_e(x) \rightarrow \neg U_e(x)$

Elimination Theorem: can eliminate fragment e if no input x that both reaches e and satisfies Δ(x); ∄x: Re(x) ∧ Δ(x)
Elimination query: Re(x) ∧ Δ(x)
- Proof: Assuming Δ(x) is true, e must be unreachable

Algorithm: first removes unreachable fragments without the well-defined program assumption; then report & remove fragments that become unreachable with this assumption (unstable code)
Simplification Theorem: can simplify e to e' if there is no input x that evaluates e(x) and e'(x) to different values, while both reaching e and satisfying the well-defined program assumption Δ(x); ∃e'∄x: e(x) ≠ e'(x) ∧ Re(x) ∧ Δ(x)
Simplification query: e(x) ≠ e'(x) ∧ Re(x) ∧ Δ(x)
oracle to propose e': Boolean oracle (true/false), Algebra oracle (eliminate common terms on both sides)
Algorithm: first replace unreachable or replaceable fragments (e(x) ≠ e'(x) ∧ Re(x) is UNSAT) without the well-defined program assumption; then report & replace fragments that become unreachable with this assumption (unstable code)

Stack does not model: memory aliasing, data races
Stack design: C: Compiler frontend; IR: UB condition insertion, Solver-based optimization, Bug report generation
Challenges: scalability, avoid false warnings by compliers, e.g. macros & inlined functions
Complier frontend: invokes clang to compile C to LLVM IR
- Ignore compiler-generated code by tracking code origins

UB condition insertion: Stack inserts a special function call for each UB condition to compute Δ(x); void bug_on(bool expr)
Solver-based optimization: elimination -> simplification with the boolean oracle -> simplification with the algebra oracle
- consults the Boolector solver to decide satisfiability

Approximate queries: computed limited to single func
- **R'$_e$(x)**: fragment e's reachability condition from the start of current function
- **dom(e)**: e's dominators: set of fragments that every execution path reaching e must have reached

Stack eliminates fragment e if following query is unsatisfiable:
- $R'_e(x) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(x)$

Stack simplifies e into e' if following query is unsatisfiable:
- $e(x) \neq e'(x) \wedge R'_e(x) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(x)$

Reachability: computed using Tu and Padua's algorithm
UB condition: collects from the bug_on calls within dom(e)
Bug report generation: reports the minimal set of UB conditions that make each report's code unstable
Greedy algorithm:
- $Q_e = H \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(x)$
- H denotes terms excluding $\bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(x)$ in Q_e
- Stack masks out each UB condition in e's dominators from Q_e individually to form a new query Q'$_e$; if the new query Q'$_e$ becomes satisfiable, then the UB condition masked out is crucial for making fragment e unstable.

Limitations: misses violations of strict aliasing and uses of uninitialized variables, not to implement undefined behavior that occurs in the frontend, may miss unstable code due to approximations, miss unstable code if the solver times out, reports false warnings when it flags redundant code as unstable
Non-optimization bugs: unstable code that causes problems even without optimization, e.g. nullptr dereference, $-2^{63}/-1$
Urgent optimization bugs: unstable code that existing compilers already optimize to cause problems, e.g. data + size < data
Time bombs: unstable code that is harmless at present, since no compiler can currently optimize it, but situation may change
Redundant code
Performance: timeout = 5s; a small number of solver timeouts due to complex reachability conditions
Unstable code caused by multiple UB: hard to manually find
Completeness: identified unstable code 7 out of 10, missed 3 b/c not implementing some UB & approx. reachability cond.
Evaluation: KLEE often fail to model undefined behavior correctly; Avoid overly aggressive compiler optimizations, e.g. by generate efficient overflow checking code; language design

DSE

DSE: Program analysis technique for automatically exploring paths through a program
Basic Idea: Run program on symbolic input, whose initial value is anything -> Program instructions become operations on symbolic expressions -> At conditionals that use symbolic inputs, fork execution and follow -> When a path terminates, generate a test case by solving the constraints on that path
SymEx examples: KLEE, CREST, SPF, FuzzBall
KLEE: Flexible symbolic execution tool based on the LLVM framework and modern SMT solvers (STP, Z3, Boolector, etc.), primarily for C code (but also C++, Rust, etc.)
Execution paths of a program: can be seen as a binary execution tree; Internal nodes are decision points in program; Leaves are program exit points; Each path from the root to a leaf represents the execution of an equivalent set of inputs
Path constraints (PC): the conjunction of constraints gathered on an execution path
Feasible paths: Symbolic execution explores only feasible paths
Implicit checks: Pointer dereferences, Array indexing, Division/modulo operations, Assert statements
All-value checks: errors found if any buggy values exist on path
Manual testing: expensive but has stronger oracles
Random Testing: ineffective if lack domain-specific guidance
Dynamic analysis: need test case; can't have all possible values
Static analysis: imprecise; hard to find bugs dependent on specific values and/or memory layout; doesn't generate test cases; usually finds more bugs; easier to apply
DSE: automatic; systematic (high cov.); reason all possible values; including bugs depending on specific values and/or memory layout; generates concrete test cases
- Bugs found w/ DSE: Expressions of Death; Disk of Death

Mixed Concrete/Symbolic Execution: can interact w/ environment, only relevant code executed symbolically
Concolic Execution: Start w/ a concrete input -> Run it, collecting PC -> Choose a PC prefix, negate last constraint -> Solve new PC to obtain a new concrete input -> Repeat
- aka directed automatic random testing / whitebox fuzzing

Non-Concolic DSE: EXE and KLEE
- PC -> Constraint solver (STP) -> generate input

Dynamic analysis: run program and observe execution
- e.g. buffer overflow detection tool intercepts memory accesses, Allocations and deallocations, Pointer arithmetic

Source-level instrumentation: easy to instrument
Binary-level instrumentation: no source or recompiling needed, no view of the stack so may not detect the overflow
Static instrumentation: change code before it is run, generate new binary, run it; Dynamic Performance, Ease of implementing
Dynamic instrumentation: instrument programs as it runs, like an interpreter; Tracking dependencies; Self-modifying code
EXE: Static instrumentation @ source-level
exe-cc: x = y -> sym[&x] = pointer if symbolic or NULL if concrete
exe-cc: v = x OP y -> sym_exp(OP, Sx, Sy) or ct(x)
exe-cc: if (x) s1; else s2 -> fork and kill path if UNSAT
All other cases can be reduced to the cases above
KLEE: Dynamic instrumentation @ intermediate level (LLVM)
- mixed concrete/symbolic interpreter for LLVM bitcode

Path Explosion

Path Explosion: real-programs have a huge number of paths
DFS: Quickly get a deep path; Not much diversity
BFS: Good diversity; shallow, Need much more memory
EXE's best-first heuristic: Pick process at the line of code run the fewest number of times, run in DFS for a while, then iterate
KLEE's MD2U heuristic: minimum distance from state s to an uncovered instruction; selection weight = 1 / MD2U(s)^2
Random Path Selection: subtrees have equal prob. of being selected; not uniform random state selection; Favors paths high in the tree; Avoid starvation
KLEE: use multiple heuristics in a round-robin fashion
Eliminating redundant paths: If two paths reach the same program point with the same constraint sets, we can prune one of them; can discard from the constraint sets of each path those constraints involving memory which is never read again
CacheSet(P) = set of reduced PCs w/ which P was visited before
ReadSet(P, PC) = memory locations read from P when reached with PC
TransCl(PC, RS) = transitive closure of all constraints in PC that overlap with the readset RS
Algorithm: When program point P reached with PC:

- Forall PC_a ∈ CacheSet(P):
  - If TransCl(PC, ReadSet(P, PC_a)) = PC_a:
    - Then stop exploration
- Explore execution tree rooted at P in DFS mode:
  - Compute ReadSet(P, PC)
- CacheSet(P) = CacheSet(P) ∪ TransCl{PC, ReadSet(P,PC)}

Redundant path elimination: non-redundant explored states & branch coverage increase faster
Statically merging paths: Phi-Node Folding (no side effects)

- max = select(a>b, a, b)
- No. paths: default = 2^N -> Phi-node folding = 1
- outsourcing problem to constraint solver

Regression Suites: most apps come with manually-written suite

- Pros: designed to execute interesting paths, good coverage of different program features
- Cons: execute each path with a single set of inputs; general case of program feature, missing corner cases

ZESTI: Using Existing Regression Suites

- Use the paths executed by the regression suite to bootstrap the exploration process
- Incrementally explore paths around dangerous operations
- No need to construct a test driver

Multipath Analysis: Bounded symbolic execution around divergence points of sensitive instructions
Other techniques: under-constrained execution, speculatively skipping code fragments

Constraint Solving

Accuracy challenge: need bit-level modeling of memory

- Systems code often observes the same bytes in different ways; Bugs often triggered by corner cases related to pointer/integer casting

Performance challenge: real programs generate many expensive constraints
Bit-level Modeling of Memory: at SMT level; theory of BV
Mirror the C type system: represent memory block by an array of 8-bit BVs; Bind types to expressions, not bits
Theory of array axioms:

- $\forall a,i,j.\ i{=}j \rightarrow \text{read}(a,i) = \text{read}(a,j)$ (array congruence)
- $\forall a,i,j,v.\ i{=}j \rightarrow \text{read}(\text{write}(a,i,v),j) = v$ (read-over-write 1)
- $\forall a,i,j,v.\ \neg(i{=}j) \rightarrow \text{read}(\text{write}(a,i,v),j) = \text{read}(a,j)$ (read-over-write 2)

Standard approach: eager translation to equi-satisfiable SAT formula; e.g. simplifying x – x to 0
Each arithmetic operation on BVs can be encoded as a circuit
Eliminate writes: $\text{read}(\text{write}(a,i,v),\ j) \leftrightarrow \text{ite}(i = j, v, \text{read}(A,j))$

- A write by itself (not inside a read) can be discarded

Eliminate reads: replace each syntactically-unique read by a fresh variable, e.g. (a[i_1]=e_1)∧(a[i_2]=e_2)∧(i_1+i_2=3) to (v_1=e_1)∧(v_2=e_2)∧(i_1+i_2=3) and (i_1=i_2 -> v_1=v_2)

- Expensive: expands each formula by n·(n-1)/2 terms
- Approximation UNSAT -> Original formula UNSAT

Array-based Refinement: when unsuccessful, STP finds an array index violating an axiom & adds all axioms involving the index
Constraint Solving: NP-complete; invoked at every branch; take most of time for many benchmarks
Constraint Solving Optimizations: Simplifying & normalizing & caching constraints, Eliminating irrelevant constraints, Exploiting subset/superset relations, Accelerating array constraints, Performing interval analysis, Summarizing loops
Eliminating Irrelevant Constraints: Eliminating constraints cannot invalidate solution; Adding constraints often doesn't invalidate solution
Irrelevant constraint elimination + caching: Significant Speedup

- Get more cache hits

Program transformations, e.g. compiler optimizations, sometimes slow down symbolic execution analysis

- **Index-based Array Transformation**
- **Value-based Array Transformation**: to nested ITE()

Applications

Under-approximation: choose concrete parameters before calling into environment -> (1) missing paths, (2) potential interaction between paths
Over-approximation: make results from environment unconstrained -> infeasible paths
Symbolic Environment: e.g. symbolic file system; virtualizes access to native resource; implemented at the OS level; no make_symbolic() calls
Environmental Modeling: Users can extend/replace environment w/o any knowledge of the tool's internals

- effective support for symbolic command line arguments, files, links, pipes, ttys, environment vars

Usage Scenarios: Automatically generate high-coverage test suites, Discover generic bugs and security vulnerabilities in complex software, Enhance the quality of regression testing, Perform comprehensive patch testing, Flag potential semantic bugs via crosschecking, Perform bounded verification of data-parallel optimizations, Refactoring, …
Testing Semantics-Preserving Evolution: find behavior diffs
Crosschecking 2 Software Versions: First go through one implementation, use the PC to go through the other
Differential Testing: Can find semantic errors, No need to write specifications, support constraint types like floating-point, many queries can be syntactically proved to be equivalent (*2 = <<1)
SIMD Optimizations: Can operate on multiple data concurrently
Key idea: tame path explosion by statically merging paths
FP issues: Precision, Rounding, Associativity, Distributivity, NaN
min/max not commutative nor associative: less then '<' always returns false if one of the operands is NaN
Underspecified behavior: semantic mismatches not always bugs
Developer can add assumptions: FP associativity, 0- and 0+

Sanitizers

Kernel space: Kernel; **User space**: Stack↓, File mappings (include libs)↓, Heap↑, BSS, Data, text
Stack Frame: args, RA, FP (frame pointer), Local vars (SP)
Basic Stack Exploit: Too long char* overflows the buffer; can crash app (DoS) or divert control flow
Heap-Based Attacks: e.g. overwrite function pointers in heap
Use After Free Attacks: can be reclaimed by another malloc
Other attacks: Overwrite longjmp buffers, Overwrite GOT (Global Offsets Table) and/or PLT (Procedure Linkage Table), Format string vulnerabilities
Out-of-bounds reads: can leak private data (e.g. Heartbleed for OpenSSL 64KB at a time) and divert control flow (e.g. when indexing into a function pointer array)
Code corruption attacks: Try to modify existing code in memory; modern CPUs mark code as read-only
Control-flow hijacking: Corrupt code pointers/code data
Non-control-data attacks: Corrupt any security-critical data
Leak confidential memory: find critical info to conduct attacks
Runtime Defenses (typically insufficient): Prevent control data from being modified, Prevent injected code from being executed, Detect invalid control-flow transfers
Stack Canaries: Add canaries to stack frames and verify their integrity prior to function return; Need to recompile code, but no source modifications needed & binary-compatible with existing libs; small performance overhead
Stack layout: args, RA, FP, Canary, Local vars
Terminator canaries: can terminates strcpy; attack: memcpy
Random canaries: value chosen at load time; attack: overwrite p to point to address of RA
Random XOR: Canary = Rand-val XOR RA; if RA was hacked, the check will fail; attack: overwrite global variable holding random value -> allocate canary table in separate R/O page; Nginx attack: overwrites stack canary 1 byte at a time
Compiler Sanitizers: dynamic analyses for finding memory safety violations and other types of undefined behavior

- static instrumentation to do runtime checks

Address Sanitizer (ASan, -fsanitize=address): detects out-of-bounds accesses, use-after-free errors, memory leaks, etc. Typical slowdown 2x.
Memory Sanitizer (MSan, -fsanitize=memory): detects uninitialized memory accesses. Typical slowdown 3x.
Undefined Behavior Sanitizer (UBSan, fsanitize=undefined): detects various types of undefined behavior. Typical slowdown 2-3x, but less data available.
ThreadSanitizer (TSan, -fsanitize=thread): detects data races and deadlocks. Typical runtime slowdown 5-15x.
Valgrind: interpreter for its own intermediate representation; works on binaries; dynamic instrumentation; approx. 10x slower
Asan: Instrument all allocation sites to add red zones (default heap redzones: 128 bytes) around memory objects. Mark those red zones as poisoned; Instrument all free() calls sites to put freed memory in quarantine and mark it as poisoned; For each memory access at *addr, add before if (Poisoned(addr)) Error();

- **Heap**: Provide own malloc version (and free)
- **Stack**: Generate code that adds redzones at runtime
- **Globals**: Redzones created at compile time

Quarantine: finding use-after-free errors
On a free, instead of returning memory to the allocator, it is placed in a quarantine; quarantined memory accessed -> use-after-free error; FIFO queue (ASan: default 256MB)
Shadow Memory:

- All bytes are addressable: shadow_val = 0
- No bytes are addressable: shadow_val < 0
- First k bytes (1 <= k <= 7) addressable: shadow_val = k

shadow_addr = offset + (addr >> 3)

- 32-bit: offset = 0x20000000 (2^29)
- 64-bit: offset = 0x0000100000000000 (2^44)

Layout: Memory, Shadow, Bad (inaccessible), Shadow, Memory
Lowest address -> first shadow byte -> first bad byte
Highest address -> last shadow byte -> last bad byte
Efficient Checks:

- shadow_addr = offset + (addr >> 3);
- shadow_val = *shadow_addr;
- if (shadow_val != 0) { // slow path
  - last_byte = (addr & 7) + access_size - 1;
  - if (last_byte >= shadow_val) Error();
- }

False Positives: None by design
False Negatives: Overflows bypassing red zones, Use-after-free with large allocations in-between drains the quarantine queue, Unaligned accesses partially out-of-bounds, Custom allocators
Msan: Detection of Uninitialized Accesses

- Shadow each allocated bit with validity bit V[b]
- uninitialized memory is allocated, V[b] = 1
- x = 1 -> V[x] = 0; x = y -> V[x] = V[y]
- Before each read b that can affect the program's observable behavior check that the b was initialized

False positives: from the interaction with uninstrumented code (Big weakness compared to Valgrind)
UBSan: Undefined Behavior Sanitizer; find e.g. Division by zero, Overshifts, Signed integer overflow, Indirect call of a function through a function pointer of the wrong type
Valgrind: cannot find stack overflow
ASan, MSan and Tsan: can detect use after free
Asan: can detect access to local variable of an ended function call, buffer overflow, Memory leaks

Data Flow

Sound static analysis: never report false negative, can infer conservative facts
Data-flow Analysis: derives information about the flow of data along program execution paths
A data-flow analysis framework (D, V, F, ∧) consists of:

- A direction of the data-flow: forwards or backwards
- A domain of abstract values V
- A family of transfer functions F: V -> V
- A meet operator ∧ : V x V -> V
- s.t. V, ∧ form a semi-lattice, framework is monotone

Reaching definitions: for each variable use, find out which definitions it may depend on

- A definition def(x) **reaches** a use use(x) if there is a path from def(x) to use(x) s.t. def(x) is not killed on that path
- A definition def(x) is **killed** on a path if there is another definition def2(x) on that path

Fixed-point Algorithm:

- OUT[ENTRY] = ∅ // boundary condition
- OUT[B] = ∅, all other B // initialization
- while (any changes in IN/OUT occur) {
  - foreach B ≠ ENTRY {
    - IN[B] = UP=predecessor(B) OUT[P]
    - OUT[B] = genB U (IN[B] \ killB)
  - }
- }

Use-def chain: for each use(x), a list of definitions reaching it
Aliases: has to conservatively reason via points-to analysis
Use Cases: Constant propagation, Detecting uninitialized accesses (create dummy definitions for all in entry block)
Live-variable analysis: find out which variables may be live at each point in the program;

- A variable is **live** at point p if it may be used along some path starting at p, without being re-defined first

Use Cases: register allocation (If value in reg not live at the end, not necessary to save; if all regs full, pick reg holding value not live), eliminating redundant paths in SymEx for constraints involving only memory locations not used anymore
Available expressions: determine expressions already computed at each program point p

- An expression x+y is available at point p if every path from the entry node to p evaluates x+y and after each evaluation there is no assignment to x or y; if x or y are redefined, a future use of x+y will need to recalculate
- A block generates {x+y} if it computes x+y and does not subsequently define x or y

Use Cases: common subexpression elimination

	Reaching definitions	**Live variables**	**Available expressions**
Direction	Forwards	Backwards	Forwards
Abstract domain	Sets of definitions	Sets of variables	Sets of expressions
Transfer functions	genB U (X \ killB)	genB U (X \ killB)	e_genB U (X \ e_killB)
Meet operator	U	U	∩
Boundary condition	OUT[ENTRY] = ∅	IN[EXIT] = ∅	OUT[ENTRY] = ∅
Initialization	OUT[B] = ∅, all other B	IN[B] = ∅, all other B	OUT[B] = U, all other B
Equations	OUT[B] = f(IN[B]); IN[B] = $\wedge_{P=pred(B)}$ OUT[P]	IN[B] = f(OUT[B]); OUT[B] = $\wedge_{S=succ(B)}$ IN[S]	OUT[B] = f(IN[B]); IN[B] = $\wedge_{P=pred(B)}$ OUT[P]

Semilattice S = (V, ∧)

- A set of values V and a meet operator ∧ : V x V -> V
- Idempotent: x ∧ x = x; Commutative: x ∧ y = y ∧ x; Associative: x ∧ (y ∧ z) = (x ∧ y) ∧ z
- Example: ∪, ∩, non-example: +

Partial order of a semilattice ≤ : x ≤ y if and only if x ∧ y = x

- x ≤ x (reflexive); If x ≤ y and y ≤ x then x = y (antisymmetric); If x ≤ y and y ≤ z then x ≤ z (transitive)
- (V, ≤) is called a poset (partially ordered set)
- A semilattice has a top element T: forall x. T ∧ x = x
- May also have a bottom element ⊥: forall x. ⊥ ∧ x = ⊥

e.g. ∧ = ∪, ≤ = ⊇, Top element: {}, Bottom element: U, Sets with more elements are considered smaller
e.g. ∧ = ∩, ≤ = ⊆, Top element: U, Bottom element: {}, Sets with more elements are considered bigger
Height of Semilattice: no. elements involved in the ascending chain - 1, e.g. height of reaching definitions semilattice with n definitions = n
Transfer Functions: F has an identity function I such that I(x) = x for all x in V; F is closed under composition: h(x) = g(f(x)) is in F
Proof of Convergence: If the framework is monotone and the semilattice has finite height, then the fixed-point algorithm is guaranteed to converge to a solution of the data-flow equations
Parametrizable by: (D, V, F, ∧) and a boundary condition

Coverage criteria and Mutation Testing

Coverage criteria: measure progress of a testing/fuzzing campaign; assess the quality of a test suite; when to stop testing
Blackbox Coverage Criteria: Based on specification

- **Pros**: for code is unavailable, can be done by users, can uncover unimplemented parts of the specification
- **Cons**: May completely miss important parts of the code

Whitebox Coverage Criteria: Based on the code
Ordered by increasing cost: Function coverage < Instruction/statement/line coverage < Branch/decision coverage < Path coverage

- deal with loops: e.g., consider only 0 or 1 iterations

100% Function coverage: No unreachable functions
100% Instruction/statement coverage: No unreachable code
100% Branch/decision coverage: No dead branches
100% Path coverage: Verification (full or bounded)
Logic-based Coverage Criteria: Criteria concerned with covering logical predicates
Branch/decision coverage: Same as for control-flow coverage
Condition coverage: atomic clause in a logical predicate

- No. tests depends on whether the logical operators have short-circuiting semantics or not

Multiple condition coverage: Percentage of combinations of condition outcomes exercised
Modified condition/decision coverage (MC/DC): Decision coverage + Condition coverage + Demonstrate that each atomic condition independently influences the branch outcome
Data-flow Coverage Criteria: how many def-use pairs covered
Mutant: small modification of program P to obtain P'
Goal of mutation testing: have a test that "kills the mutants"
Competent Programmer Hypothesis: programmers tend to write programs close to the correct version. Errors are simple and can be corrected by simple syntactical changes.
Coupling Effect Hypothesis: A test suite that detects simple faults can also detect complex ones.
Mutation: Constant replacement, Operator replacement, Array index replacement, Variable replacement, Statement deletion
First-order mutant: apply a mutator operator once
Higher-order mutant: apply a mutator operator multiple times
Strongly killed: a test leads to different program outputs in P and P'
Weakly killed: a test leads to different program states (or more generally for a component to compute different values)
Mutation coverage / score = # killed mutants / # total mutants
Surviving mutants: mutants which are not killed
Challenges: Mutant construction, Performance, Equivalent mutant problem (to original program; wastes developers' time), Indistinguishable mutant problem (to each other; distort the mutation score)
Caution: test suite with 100% coverage can still miss bugs