Trustworthy Software Systems / 2026 / Checkpoint-1
Name, student-id_____

1) [4 pts] KLEE explores multiple execution paths simultaneously by maintaining a separate state per path. What is the primary resource overhead of this approach at scale? Briefly describe how KLEE keeps it tractable.

> The dominant cost is memory: each live path requires its own copy of the program's heap and accumulated constraints, and the number of live paths grows exponentially with the number of symbolic branches.
> KLEE mitigates this using copy-on-write memory at the object level. States initially share memory objects, and when a state writes to one, KLEE creates a private copy only for that state. This avoids duplicating large portions of memory that remain unchanged across paths.

2) [4 pts] KLEE models the OS environment rather than running against a real one. Why is this necessary, and what does it cost? Give a concrete example of a bug that falls outside the scope of KLEE.

> It is necessary because real system calls require concrete inputs — passing a symbolic file descriptor or a symbolic buffer pointer to the kernel will fail immediately. Instead, to keep values symbolic across call boundaries, KLEE intercepts system calls and returns modeled responses (e.g., a symbolic byte array in the place of a real file read). Without this, symbolic execution would collapse to concrete execution at the first syscall, which would entirely sacrifice completeness.
> Any syscall-related bug falls out of the scope of KLEE.

3) [4 pts] How many symbolic states must KLEE maintain after the execution of the following branch?

```
klee_make_symbolic(x, sizeof(x), "x");

for (int i = 0; i < 100; i++)
   if (x[i] > 0)
       foo();
    else
        bar();
```

> In this program, the condition x[i] > 0 depends on symbolic input. KLEE forks execution at each iteration of the loop, creating two states for every branch. After n iterations, there are 2^n = 2^100 execution paths.

4) [8 pts] KLEE uses two heuristics to choose which symbolic state to execute next: (a) random-path selection, and (b) coverage-optimized selection. What problem does each solve, and why is neither sufficient alone?

> Random-path selection picks uniformly among all live states to solve the starvation problem: a long loop or deep recursion could dominate CPU time, starving every other path. Coverage-optimized selection prefers states whose nearest uncovered branch is closest. It solves the redundancy problem: random selection tends to re-execute already-covered code. Neither alone is sufficient — random selection wastes time re-covering the same lines; coverage-optimized gets trapped in dense, hard-to-cover regions and can starve out entire subtrees it never reaches.

5) [8 pts] *SAGE* operates at the x86 instruction level. *Give a concrete advantage of doing so over KLEE, beyond the obvious "no source is required."*

> SAGE analyzes the actual deployed artifacts along with their environment, including closed-source third-party DLLs and OS libraries that ship with the product.
> A source-level tool like KLEE would need IR for every library—unavailable for proprietary components.

SAGE operates at the x86 binary level through a pipeline of three concrete components:

1. **Concrete execution with instruction-level tracing (iDNA/Nirvana):** SAGE first runs the target program natively on Windows — real process, DLLs, syscalls, instruction-level execution trace: Every x86 instruction executed, in order, with its operand values. This is not source-level profiling — it is a full record of every machine instruction the CPU executed.
2. **Symbolic replay.** SAGE replays the recorded instruction trace through its own symbolic execution engine, processing each x86 instruction one at a time. Instructions that operate on input-derived bytes are executed symbolically — their results are symbolic expressions over the input bytes. Instructions that operate on non-input data are executed concretely using the values from the trace. A taint tracker identifies which bytes are input-derived: it marks the file's bytes as symbolic at the point they are read (e.g., via ReadFile()), and propagates the taint forward through every instruction that touches them. At each conditional branch whose condition depends on tainted bytes, the engine records a constraint. The result is a path condition expressed as bitvector arithmetic over the raw input bytes.
3. **Constraint solving.** Generational search takes the path condition and negates one constraint at a time. Each negated system is handed to an SMT solver operating over bitvectors. The solver returns a concrete byte assignment satisfying the new constraint, which becomes a new input file handed back to step-1.

6) [8 pts] Describe an input format for which *SAGE* is a poor fit in generating test inputs for, and explain why. (Hint: What info is unavailable to *SAGE*, and how does this affect its ability to simplify constraints and prune infeasible paths?)

> A structured, schema-based format, such as XML, is a poor fit because SAGE lacks domain-specific knowledge about hierarchical structure, well-formedness rules, and semantic relationships between fields. It treats inputs as raw byte sequences and reasons only at the bitvector level, without understanding constraints such as tag nesting, matching opening and closing tags, or schema-defined dependencies. As a result, it cannot exploit high-level invariants to simplify constraints or rule out structurally invalid inputs early, leading to more complex solver queries and reduced ability to efficiently prune infeasible paths.

7) [8 pts] IvySyn first discovers inputs that trigger crashes in the low-level C++ kernel implementations, and then uses those crashing inputs to synthesize Proof-of-Vulnerabilities (PoVs) in Python. Why is this necessary? Can't the test inputs crashing low-level kernel C++ implementations be used to report a bug to the corresponding developers?

> Without synthesizing PoVs, it is unclear whether the derived crashing input can indeed be generated as a result of reaching the low-level kernel via a valid path starting from the developer-facing Python level APIs, or if they are irrelevant false-positives that can not be triggered in a real, end-to-end system as opposed to an isolated, myopic view of the low-level APIs.

8) [8 pts] KLEE and SAGE both need an oracle to decide whether a test input reveals a bug. KLEE flags memory safety violations and failed assertions, and SAGE uses a runtime sanitizer to flag crashes and memory corruptions.

a) Identify a class of bugs that both tools will miss due to their shared oracle assumptions.

b) Propose an extension that would let one of these tools detect the class of bugs identified in (a).

c) What is the false positive risk of the oracle suggested in (c)?

> a) Silent data corruption — bugs where the program produces wrong output but does not crash and violates no memory safety property. Example: an integer overflow in an image decoder that silently produces a corrupted pixel value. All three tools use crash/sanitizer oracles: if the program exits cleanly, no bug is reported.

> b) Differential oracle: run two semantically equivalent implementations of the same function (e.g., a reference implementation and the target) on the same symbolic or generated inputs, and flag any output divergence. Applied to KLEE: mark inputs symbolic, run both implementations in the same KLEE instance, and assert output equality at the return site. The oracle checks the assertion; divergence is a bug.

> c) False positive risk: legitimate behavioral differences between implementations (e.g., different handling of edge cases the spec leaves undefined, different floating-point rounding) will be flagged.

9) [8 pts] IvySyn versus KLEE. a) Scalability: Which has the advantage, if any, and why? b) Precision regarding false positive rate: Which has the advantage, if any, and why? c) Bug class coverage, beyond memory safety: Which has the advantage, if any, and why?

> **IvySyn**. It treats the DL framework's native code as a black box, invoking operators through the Python API and observing crashes. It never symbolically executes the native code, so its exploration space is bounded by the number of operator signatures, not the size of the implementation. KLEE's state space explodes with program size.

> **Same for both.** Every IvySyn report comes with a PoV and KLEE respectively generates counterexamples/ test cases.

> **KLEE**. IvySyn is specifically engineered to find memory safety errors, while KLEE can discover any observable behavioral deviation expressible as an assertion or branch condition — incorrect output values, mishandled error codes, resource leaks, assertion failures — making its bug class coverage substantially broader.

10) [10 pts] An OS fuzzer targeting the Linux kernel needs seed syscall sequences. A tracer collected the following strace output from a real program:

```
socket(AF_INET, SOCK_STREAM, 0)              = 3
bind(3, {sa_family=AF_INET, port=8080}, 16)  = 0
listen(3, 5)                                 = 0
open("/etc/hosts", O_RDONLY)                 = 4
read(4, buf, 4096)                           = 312
close(4)                                     = 0
accept(3, NULL, NULL)                        = 5
write(5, buf, 312)                           = 312
close(5)                                     = 0
close(3)                                     = 0
```

- int socket(int domain, int type, int protocol); Creates an endpoint for communication and returns a descriptor.

- int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen); When a socket is created, it exists in a name space (address family) but has no address assigned to it. bind() assigns the address specified by addr to the socket referred to by sockfd.

- int listen(int sockfd, int backlog); Marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept.

- int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen); Extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket.

a) Identify all explicit dependencies in this trace. (An explicit dependency is one where a syscall produces a return value that another syscall uses directly as an argument.)

b) Identify all implicit dependencies in this trace. (An implicit dependency is when one where a syscall affects program state influencing other syscall's behavior without a direct argument link.)

c) You want to distill (i.e., minimize) this trace to the smallest possible seed that still exercises the accept() path in the kernel. Which syscalls can you safely drop, and which must you keep?

> (a) Explicit dependencies:

- bind(...) <- fd=3 <- socket(...)
- listen(...) <- fd=3 <- socket(...)
- read(...) <- fd=4 <- open(...)
- close(...) <- fd=4 <- open(...)
- accept(...) <- fd=3 <- socket(...)
- write(...) <- fd=5 / buf <- accept(...) / read(...)
- close(...) <- fd=5 <- accept(...)
- close(...) <- fd=3 <- socket(...)

> (b) Implicit dependencies:

- listen <- bind success
- accept <- listen success / bind success
- write <- read

> (c) Distilled seed to reach accept():

- accept -> [accept, listen, bind, socket]

11) [10 pts] The following code is from a hypothetical server. db_connect() returns a valid connection pointer on success, or NULL on failure (host unreachable, auth error, pool exhausted, etc.).

/* file1: utils.h */
```
#define CHECK_NULL(ptr, msg) \
    do { if (!(ptr)) { log_fatal(msg); exit(1); } } while (0)
```

/* file2: server/main.c */
```
conn = db_connect(cfg->host, cfg->port);
if (!conn) {
    log_error("main: db_connect failed");
    return -1;
}
process_requests(conn);
```

/* file3: server/worker.c */
```
conn = db_connect(cfg->host, cfg->port);
process_requests(conn);
```

/* file4: server/admin.c */
```
conn = db_connect(cfg->host, cfg->port);
if (!conn)
    return -1;
run_admin_query(conn);
```

/* file5: server/backup.c */
```
conn = db_connect(cfg->host, cfg->port);
if (!conn) {
    schedule_retry();
    return;
}
run_backup(conn);
```

/* file6: server/init.c */
```
conn = db_connect(cfg->host, cfg->port);
CHECK_NULL(conn, "init: db_connect failed");
init_schema(conn);
```

---

a) What may-belief would a cross-checker infer from this corpus? State it precisely, give its statistical confidence, and identify which sites it flags as violating it.

b) File-3 (server/worker.c) is a real bug, while file-6 (server/init.c) is not, but both would be flagged as bugs. Explain what causes the false positive at file-6?

> (a) Sites 1, 3, and 4 all follow an explicit if (!conn) guard after calling db_connect(). The cross-checker infers the may-belief: "the return value of db_connect() should be checked for NULL before use." Statistical confidence: 3/5 = 60%. Sites flagged: Site 2 and Site 5 — both call db_connect() and proceed to the next statement without a visible if (!conn) pattern.

> (b) Site 2 is genuinely unsafe: if db_connect() returns NULL, process_requests(conn) immediately dereferences a NULL pointer, crashing the server or corrupting memory. Site 5 is safe: CHECK_NULL expands to an if (!(conn)) { log_fatal(...); exit(1); } guard — the NULL case is caught and the process exits before init_schema(conn) is ever reached. The false positive arises because the checker operates on source tokens, not on program semantics. It looks for the syntactic pattern if (!conn) and never sees CHECK_NULL's expansion. From the checker's perspective, Sites 2 and 5 are structurally identical: a call to db_connect() followed immediately by use of the return value with no intervening conditional.

12) [15 pts] Describe how you would extend DART to systematically find deadlocks in multithreaded C programs.

a) What changes are needed to (i) the random phase, (ii) the symbolic phase, and (ii) the termination condition?

b) Your extended DART and Dimmunix both attempt to uncover deadlocks, but at different points in the software deployment lifecycle. Compare them on (i) when in the deployment lifecycle each tool is applied, (ii) what each tool produces as output, and (iii) what each tool requires from the developer as its input.

c) A program has a deadlock that only manifests under a specific interleaving of three threads and a specific input value. Which tool is more likely to find it, and why the other will likely miss it?

> **a-i) Random phase:** In addition to randomizing input values, also randomize thread scheduling, by injecting random sched_yield() calls at synchronization points. This explores different interleavings.

> **a-ii) Symbolic phase:** Extend path condition collection to include both data constraints and interleaving constraints, and negate interleaving constraints in addition to data constraints to generate both new inputs and new schedules.

> **a-iii) Termination:** All reachable (input × interleaving) pairs have been explored, or bound reached.

> **b-i)** Extended DART is applied during development, as a testing tool run before deployment, while Dimmunix is applied in production, as a runtime defense in deployed systems.

> **b-ii)** Extended DART produces a concrete (input + interleaving) pair that reproduces the deadlock—a bug report developers can act on; Dimmunix produces a runtime avoidance mechanism that prevents deadlock without identifying the root cause.

> **b-iii)** Extended DART requires developers to write a harness specifying which inputs are under test; Dimmunix requires no developer effort — it is transparent to the application.

> **c)** The three-thread, specific-input deadlock is more likely to be found by the extended DART. It has a concrete (input value, interleaving) pair that extended DART's symbolic phase can construct by negating the relevant data and scheduling constraints. Dimmunix will miss it until it actually occurs in production. Dimmunix, on the other hand, provides no pre-deployment detection and it only builds immunity after the first deadlock occurrence. If the deadlock is rare enough that it never manifests in production testing, Dimmunix's signature database remains empty for this pattern indefinitely.

13) [20 pts] Consider the following C function:

```
void testme() {
  char *s;
  char c; /* sizeof(char) == 1 */
  int state = 0;

  while (1) {
    c = input();
    s = input();
    if (c == 'V' && state == 0) state = 1;
    if (c == 'E' && state == 1) state = 2;
    if (c == 'Z' && state == 2) state = 3;
    if (c == 'E' && state == 3) state = 4;
    if (c == 'N' && state == 4) state = 5;
    if (c == 'K' && state == 5) state = 6;
    if (c == 'O' && state == 6) state = 7;
    if (c == 'V' && state == 7) state = 8;

    if (s[0]=='M' && s[1]=='V' && s[2]=='P' && state == 8)
      ERROR;
  }
}
```

---

a) What is the expected number of test inputs that pure random testing needs to generate in order for state to reach value 8? Ignore the last if-statement.

b) What is the expected number of test inputs that pure random testing needs to generate in order to trigger the error "ERROR" assuming state has reached the value 8?

c) What is the expected number of test inputs that a systematic test input generator, which combines concrete and symbolic execution in order to systematically explore program paths and generate new inputs by solving path constraints (e.g., DART and SAGE), needs to generate in order for state to reach value 8? Ignore the last if-statement.

d) What is the expected number of test inputs that a systematic test input generator needs to generate in order to trigger the error "ERROR" assuming state has reached the value 8?

e) Observe the numbers of (a), (b), (c), and (d) and suggest a test input generation approach that will be appropriate in uncovering errors in functions similar to "testme()".

> **a)** 8 independent geometric trials, p = 1/256 each: E = 8 × 256 = 2,048

> **b)** Need s[0]='M', s[1]='V', s[2]='P' simultaneously. Three independent bytes: E = (2^8)^3 = 16M

> **c)** The DFS must explore all possible values of c across the 8 iterations required. c is 1 byte (256 values), and each iteration's choice is independent: E = 8^8 = (2^3)^8 = 2^24 = 2^4*10^6 = 16M

> **d)** With state=8 concrete, the condition reduces to s[0]=='M' && s[1]=='V' && s[2]=='P'. Short-circuit && creates exactly 4 symbolically distinct path outcomes: fail at byte 0, fail at byte 1, fail at byte 2, or pass all three. The constraint solver reaches the ERROR in 4 test inputs. Note that at the x86 assembly level, each "&&" with an input dependency cannot be optimized by the compiler and leads to an individual conditional, which is why the answer is four and not two.

> **e)** Random is good at what concolic is terrible at, and vice versa. Hybrid uses random for the state machine (~2,048 inputs) and concolic for the string check (~4 inputs), reaching ERROR in ~2,052 total — orders of magnitude better than either alone.

> Read this paper for reference.