Trustworthy Software Systems / 2026 / Midterm Exam
Name, student-id _____

1) [30 pts] REST API Fuzzing.

Consider the following OpenAPI specification for a project-management cloud service:

> POST /users
- **Body:** { "email": string, "name": string }
- **Response:** { "user_id": integer }

> POST /users/{user_id}/projects
- **Body:** { "owner_id": integer, "name": string }
- **Response:** { "project_id": uuid }

> POST /projects/{project_id}/issues
- **Body:** { "title": string, "assignee": integer }
- **Response:** { "issue_id": integer }

> DELETE /projects/{project_id}/issues/{issue_id}
- **Body:** None
- **Response:** 204 No Content

---

a) [7 pts] Identify all producer–consumer dependencies RESTler will infer from this OpenAPI specification. (Enumerate them in a list with entries of the following format: *"Endpoint: <HTTP VERB + URI>" produces "X" of type "Y"* or *"Endpoint: <HTTP VERB + URI>" consumes "X" of type "Y"*; where "X" is a resource name, and "Y" is its type. If one endpoint is both a consumer and a producer, add two entries in the list for it.)

> Example: "Endpoint: <POST /users>" produces "user_id" of type "integer".

"Endpoint: <POST /users/{user_id}/projects>" consumes "user_id" of type "integer"

"Endpoint: <POST /users/{user_id}/projects>" produces "project_id" of type "uuid"

"Endpoint: <POST /projects/{project_id}/issues>" consumes "project_id" of type "uuid"

"Endpoint: <POST /projects/{project_id}/issues>" produces "issue_id" of type "integer"

"Endpoint: <DELETE /projects/{project_id}/issues/{issue_id}>" consumes "project_id" of type "uuid"

"Endpoint: <DELETE /projects/{project_id}/issues/{issue_id}>" consumes "issue_id" of type "integer"

b) [2 pts] Calculate the minimum sequence length required for RESTler to successfully exercise the business logic behind the API endpoint <DELETE /projects/{project_id}/issues/{issue_id}>.

>

c) [5 pts] RESTler keeps in its `*seqSet*` only sequences leading to status codes belonging to the 2xx class (e.g., *200-"OK"*; or, *201-"Created"*). i) Why is this filtering necessary? ii) Does it reduce RESTler's ability to effectively test a target cloud service? If yes, how; if not, why.

>

Filtering is necessary because keeping every sequence — including those that ended in 4xx — causes a state-space explosion: a rejected request did not actually advance the service's state, so any sequence built on it just wastes the budget exploring downstream from a broken prefix. This compromise is usually beneficial: it focuses exploration on reachable service states, where deeper bugs are more likely to live. It does lose completeness, because a non-2xx response can still have side effects or expose an interesting error path. RESTler partly handles this by logging 5xx responses as bugs before discarding the sequence. The tradeoff is pragmatic: RESTler sacrifices some paths after rejected requests to spend most of its time exercising valid stateful workflows.

d) [3 pts] You run RESTler with a limited time budget, say 5 hours, on the following three services, wishing to find as many crashes as possible, in the given time frame. For each service, choose either (i) the BFS-Fast state space exploration strategy, or (ii) the RandomWalk state space exploration strategy, or (iii) None (if neither BFS-Fast nor RandomWalk seem fit) and comment on your choice. (Answer in the format: "Service X with strategy Y, because...")

> Service A: Issue-tracking API
Example workflow: POST a fresh issue -> GET/DELETE/UPDATE an existing issue

> Service B: Deployment workflow API
Example workflow: Create org -> Create project -> Upload artifact -> Create environment -> Deploy -> Poll status

> Service C: Loan-approval API
Example workflow: Upload identity docs -> Verify identity -> Attach bank account -> Submit application -> Accept terms

---

>

- Service A: BFS-Fast, because many short create-then-consume paths can be covered systematically.

- Service B: RandomWalk, because useful behaviour is behind a strict long workflow, and exhaustive sibling expansion wastes budget on invalid prefixes.

- Service C: RandomWalk, similarly to the above, because the ordered workflow is deep.

e) [3 pts] Assume we update the user registration endpoint as shown below. For which fields in the request body does RESTler need predefined fuzzing dictionary values? And which fields do not require such a dictionary, and why?

POST /users
- **Body:** { "email": string, "name": string, "role": enum("user", "admin") }
- **Response:** { "user_id": integer }

>

RESTler needs predefined fuzzing-dictionary values for the primitive string fields: email and name. These fields have large input domains, so RESTler cannot enumerate all possible strings. It needs a dictionary of representative string values, such as empty strings, long strings, special characters, Unicode strings, malformed emails, etc. The role field does not require a general string fuzzing dictionary because its valid values are already specified by the enum. RESTler can render those values directly from the specification. If the goal is to test validation behaviour, RESTler could additionally try invalid enum values, but those would be deliberate negative tests rather than required values for normal, valid rendering.

f) [2 pts] Identify two distinct reasons why Pythia can outperform RESTler's static fuzzing dictionary, even though both produce syntactically valid inputs.

>

First, Pythia has open-vocabulary string mutations: it can generate seed-shaped but novel values such as realistic email variants, paths, or names. Second, it can mutate multiple fields in correlated ways through one latent perturbation, exploring joint ill-formedness that per-field dictionary lookup misses.

g) [8 pts] You are extending RESTler from a black-box stateful REST API fuzzer into a white-box concolic executor, akin to SAGE. The target is a RESTful cloud service backed by a database.

RESTler has already been changed so that fuzzable request fields, namely body fields, path parameters, and query parameters, are marked symbolic at request-generation time. RESTler still sends concrete REST API request sequences to the running service. Beyond marking request fields symbolic, describe what additional changes are needed so RESTler can generate new tests by exploring program paths rather than using a static fuzzing dictionary?

This is an open-ended system design question, and you may answer it at a high-level by discussing the following:

- Constraints RESTler should collect during execution: What are they, and where to collect them from?

- Use of constraints collected to generate new test cases: How?

- Instrumentation needed in the application and database layer: Can it be as generic and easy to adopt as possible?

>

Constraints collected, what and where. During each concrete run, RESTler collects a path condition: the conjunction of predicates and result-assumptions encountered as the request sequence executes. Each predicate is a relation over the symbolic request fields (e.g., email_sym contains "@", SELECT users WHERE email = email_sym); each assumption captures the outcome that determined subsequent control flow (e.g., "returned 0 rows", "returned a row with id = U"). The collection point is the database boundary, a logging proxy in front of the database, the database's own query log, or a thin wrapper around the connection driver. Every SQL statement issued while a RESTler-generated request is in flight is captured, together with its parameter bindings and result. The path condition is sequence-level, not request-level: RESTler's machinery threads dynamic objects from one request into the next, so the constraint that "the U written by request 2's INSERT is the U read by request 3's SELECT" only exists across requests, and IDOR-style or broken-state-transition bugs depend on exactly such cross-request links.

Use of constraints to generate new tests. Classical concolic loop. RESTler picks one predicate or result-assumption in the collected path condition; for example, "SELECT users WHERE email = email_sym returned 0 rows", negates it while keeping the rest fixed, hands the resulting system to an SMT solver, and asks for satisfying values. The solver's model is translated back into concrete bindings for the symbolic body / path / query slots and plugged into RESTler's existing request templates; the sequence is then re-executed. The OpenAPI grammar, request ordering, authentication context, and producer–consumer machinery are untouched — the solver only fills the symbolic slots. Each negation explores one new path; iterating drives coverage of database-visible branches the service takes because of what it sees in the database. The bug oracle is unchanged: 5xx responses and active property-checker violations remain the only signals of a bug.

Instrumentation is kept generic and easy to adopt. The deliberate design choice is to instrument only the database layer and skip the application's language runtime entirely. No taint tracking inside controllers, validators, or ORMs; just a SQL-log tap. Symbolic taint is recovered at the database boundary by value matching: each symbolic field has a known concrete value at the point of issue, and any parameter or literal appearing in the captured query log that equals that value is treated as derived from the corresponding symbolic input. The query then becomes a predicate over the symbolic input, and the database's response becomes the result-assumption. This makes the executor stack-agnostic — any high-level MVC framework with a SQL backend works without source modification — and reduces deployment to a single proxy or driver shim. The trade-off is completeness: branches that never reach the database — pure input validation, in-memory session checks, cache-only reads, feature flags — are invisible to the executor. Their conditions still execute concretely but cannot be flipped. The class of bugs that the design does capture – those depending on the interaction between the request and the persistent state of the database, including the cross-request producer–consumer flows that black-box RESTler cannot reason about — is exactly the class where REST API bugs most often hide, which is why the simpler design is a sensible engineering choice.

2) [30 pts] ACIDRain and a RESTler-Style Concurrency Checker.

Assume an e-commerce cloud store whose database has an initial state: <item 42 has stock = 1>, and users A and B both have enough balance to buy it. Assume, also, that the database isolation level is READ COMMITTED (i.e., each SQL statement sees only data committed before that statement begins, but later statements may observe a different database state) and AUTOCOMMIT is ON (i.e., unless the programmer explicitly opens a transaction, each SQL statement runs as its own transaction). Two concurrent calls hit with *buy(42, A)* and *buy(42, B), where:*

 # Variant 1

```
def buy(item_id, user_id):
    stock = SELECT stock FROM items WHERE id = item_id
    if stock <= 0:
        return "out of stock"
    price = SELECT price FROM items WHERE id = item_id
    balance = SELECT balance FROM users WHERE id = user_id
    if balance < price:
        return "insufficient funds"
    UPDATE items SET stock = stock - 1 WHERE id = item_id
    UPDATE users SET balance = balance - price WHERE id = user_id
    INSERT INTO orders(item_id, user_id)
    return "ok"
```

 # Variant 2

```
def buy(item_id, user_id):
    BEGIN
    stock = SELECT stock FROM items WHERE id = item_id
    if stock <= 0:
        ROLLBACK
        return "out of stock"
    price = SELECT price FROM items WHERE id = item_id
    balance = SELECT balance FROM users WHERE id = user_id
    if balance < price:
        ROLLBACK
        return "insufficient funds"
    UPDATE items SET stock = stock - 1 WHERE id = item_id
    UPDATE users SET balance = balance - price WHERE id = user_id
    INSERT INTO orders(item_id, user_id)
    COMMIT
    return "ok"
```

---

a) [10 pts] Define a scope-based anomaly and a level-based anomaly, and describe what anomaly (if any) each of the two buy() variants above has, and why.

>

A scope-based anomaly occurs when the application puts the wrong operations inside a transaction. Variant 1 is scope-based: with AUTOCOMMIT on and no BEGIN, every SQL statement in buy() runs as its own transaction, so the six reads and writes that together implement "buy item" (stock read, price read, balance read, stock update, debit, order insert) are not atomic. A level-based anomaly occurs when the transaction scope is right, but the isolation level is too weak. Variant 2 illustrates this: the whole buy() operation is in one transaction, but READ COMMITTED may still allow a non-serializable interleaving.

b) [5 pts] For Variant-1, use the format <T1/2: "SQL statement" -> Value> (e.g., T1: "SELECT stock …" -> 1), to show an interleaving of two threads where both purchases succeed. What invariant is violated?

>

T1: SELECT stock -> 1
T2: SELECT stock -> 1
T1: SELECT price, SELECT balance(A) -> enough balance
T2: SELECT price, SELECT balance(B) -> enough balance
T1: UPDATE stock = stock - 1; debit A; INSERT order
T2: UPDATE stock = stock - 1; debit B; INSERT order

Both purchases succeeded even though only one unit was in stock. The violated invariant is that successful orders for an item must not exceed the item's available stock; equivalently, stock should not become negative.

c) [5 pts] For Variant-2, is the application safe from overselling? (If not, show a bad interleaving and classify the anomaly.)

>

No. At READ COMMITTED, both transactions can still make the purchase decision from the same old stock value:

T1: BEGIN
T2: BEGIN
T1: SELECT stock -> 1
T2: SELECT stock -> 1
T1: SELECT price, SELECT balance(A) -> enough balance
T2: SELECT price, SELECT balance(B) -> enough balance
T1: UPDATE stock = stock - 1; debit A; INSERT order; COMMIT
T2: UPDATE stock = stock - 1; debit B; INSERT order; COMMIT

This is a level-based anomaly. The transaction scope is now correct, but READ COMMITTED does not force the execution to match any serial order. In a serial execution, one buyer would succeed first, and the second would then see stock = 0.

d) [10 pts] Suppose the e-commerce cloud store exposes the function buy() via the following REST APIs:

> POST /items/{item_id}/buy: Attempts to buy a unit of an item with "item_id" for the user with "user_id".
  - Body: { "user_id": integer }

There are also the following read APIs:

> GET /items: Returns a list of all item ids currently in stock

> GET /items/{item_id}: Returns the current item state, including its stock

> GET /users: Returns a list of all user ids belonging to signed-up users

> GET /orders?user_id={user_id}: Returns the orders successfully placed by the given user

Show a sequence of API requests that, if derived (i.e., if automatically generated by a tool), can be used to test for and detect overselling bugs. Then, design a RESTler-II-style concurrency active anomaly checker with the goal of uncovering such overselling bugs. (Assume appropriate registered users already exist, and thus, the <GET /users> API will return valid user ids, if invoked.) The latter is an open-ended system design question, and you may answer it at a high-level by discussing the following:

- What RESTler sequence prefix should the active checker build upon? And how?
- What follow-up checks should it perform, and what oracle should it use?

>

The checker reuses a prefix RESTler has already exercised in seqSet that ends with both POST /users having succeeded and a POST that yields an item with stock = 1 (or the checker seeds it directly via the OpenAPI grammar). It records the item's initial stock via GET /items/{item_id} as the precondition for the oracle.

It records the initial item state:

GET /items/{item_id}

Then it issues two concurrent requests:

User A: POST /items/{item_id}/buy
User B: POST /items/{item_id}/buy

Afterwards, it checks:

GET /orders?user_id=A
GET /orders?user_id=B

The oracle is: if the initial item had stock = 1, both users' order lists must not contain an order for the same item_id. If both do, the checker reports a concurrency anomaly.

3) [40 pts] Speculative execution attacks.

Read carefully the modified Spectre-v1 proof-of-concept, which appears below:

```
/*
 * _mm_clflush(p): Evicts the cache line containing address p from all cache levels.
 * __rdtscp(&out): Reads the CPU's 64-bit cycle counter (TSC) and returns it. (&out receives an unused core-id value.)
 */
static int       results[16];
unsigned int   array1_size = 24;
uint8_t        array1[192] = {7,11,3,17,29,5,13,19,2,23,31,37, 41,43,47,53,59,61,67,71,73,79,83,89};
uint8_t        array2[16 * 1024], temp = 0;

void victim_function(size_t x) {
    if (x < array1_size) {
       uint8_t v = array1[x] & 0x0F;
       temp ^= array2[v * 1024];
   }
 }
 void readMemoryByte(size_t malicious_x) {
    const int          CACHE_HIT_THRESHOLD = 100;
    size_t             training_x, x;
    int                i, j, junk = 0;
    uint64_t           t0, t1;
    volatile uint8_t  *addr;

    for (i = 0; i < 16; i++) results[i] = 0;

    for (int tries = 1000; tries > 0; tries--) {

       for (i = 0; i < 16; i++)
          _mm_clflush( ____________________________________ ) ;   /* (1) FILL ME */

       training_x = tries % array1_size;
       for (j = 100; j >= 0; j--) {
          _mm_clflush( ____________________________________ ) ;  /* (2) FILL ME */

          for (volatile int z = 0; z < 120; z++) {}
          if (j % 7 == 0)
             x = ____________________________________ ;  /* (3) FILL ME */
          else
             x = ____________________________________ ;  /* (4) FILL ME */

          victim_function(x);
      }
       for (i = 0; i < 16; i++) {
          addr = ____________________________________ ; /* (5) FILL ME */
          t0   = __rdtscp(&junk);
          junk = ____________________________________ ; /* (6) FILL ME */
          t1   = __rdtscp(&junk) - t0;
          if (t1 <= CACHE_HIT_THRESHOLD && ____________________________________)  /* (7) FILL ME */
             results[i]++;
      }
   }
}
```

a) [15 pts] Fill in the seven blanks in the modified Spectre-v1 proof-of-concept above, and give a short sentence explaining what role each line plays in the attack. (Fill the missing lines in place and comment below using the format "(1) Line does X and is needed because Y…")

>

(1) Line evicts every probe address of array2 from the cache, and is needed because the timing loop later distinguishes "secret-warmed" buckers from cold ones; without a clean baseline, buckets left hot by previous iterations or by hardware prefetching would register as false positives and corrupt the score.
(2) Line evicts the bounds-check operand array1_size, and is needed because the comparator now has to fetch it from DRAM (~250 cycles); that stall is precisely the speculation window during which the predictor-driven path issues array1[x] and the dependent probe load and leaves a cache footprint before the misprediction is detected.

(3) Line passes the out-of-bounds attacker-controlled index on every 7th iteration, and is needed because this is the call that rides the predictor's "enter the body" history into a transient read of forbidden memory — without it, there is no leak, only training.
(4) Line passes a small in-bounds index on the other 6/7 iterations, and is needed because repeatedly exercising the true branch is what teaches the global branch predictor that this branch enters the if-body, so that the predictor still says "taken" when the malicious out-of-bounds call arrives, and the comparator has not yet resolved.
(5) Line points to the i-th mailbox of array2, and is needed because the probe must use the same 1024-byte stride the victim used to encode v — each iteration of this loop tests exactly one candidate value of the secret; a different base or smaller stride would either probe the wrong buffer or conflate adjacent secret values into one cache line.
(6) Line performs the load whose latency is being measured, and is needed because it is the access whose hit/miss distinguishes array2[i*1024] between L1 (~4 cycles) and DRAM (~200+ cycles); the two __rdtscp calls bracket this load, so omitting it leaves nothing to time.
(7) Line excludes the mailbox that legitimate training has warmed, and is needed because the in-bounds calls really do execute the if-body and warm array2[(array1[training_x] & 0x0F) * 1024] on every iteration; if that mailbox were counted, public training data would dominate results[] and the attack would report array1[training_x] instead of the actual secret half-byte.

b) [10 pts] What capability does readMemoryByte, paired with victim_function, give an attacker? Discuss the kind of primitive (read or write), its granularity, and the range of memory it can cover.

>

It is an arbitrary-read primitive at 4-bit (half-byte) granularity, read-only, no architectural side effects in the victim, no fault. Per call, the attacker picks any address (via malicious_x) and recovers 4 of the 8 bits stored there: address reach is unrestricted, but resolution is half a byte. The reachable range is every byte mapped into the address space of the process executing the gadget — heap, stack, BSS, dynamic libraries, JIT regions, and on pre-KPTI systems, the kernel's directly-mapped memory. In practice, this exposes anything the victim keeps in memory (keys, session tokens, ASLR-randomised pointers, inter-tenant data in shared runtimes).

c) **[10 pts] You are scanning the binary for a second gadget that, combined with victim_function, would let you recover full bytes from any address. You found the function below:**

```
void victim_function2(size_t x) {
    if (x < array1_size) {
        uint8_t v = array1[x] >> 4;
        temp ^= array2[v * 1024];
    }
}
```

Why is this exactly what you need? Explain how you would change the function readMemoryByte from (3-a) to also use the victim_function2 discovered above?

> victim_function masks with & 0x0F, so the cache index is driven only by the low nibble $b_3..b_0$ of array1[x]; the high nibble is zeroed before it ever influences any address and is therefore invisible to the cache channel. victim_function2 shifts with >> 4, which moves the high nibble $b_7..b_4$ into bits 0..3 of v, where it now drives the same v * 1024 cache index. The two gadgets are complementary: between them, they cover all 8 bits of every byte in the program's address space: Gadget 1 carries the low half; Gadget 2 carries the high half.

What changes in readMemoryByte? Almost nothing. Two edits:

- The call site victim_function(x) becomes victim_function2(x).
- The training-noise filter in blank (7) becomes **i != (array1[tries % array1_size] >> 4)** so it subtracts the mailbox warmed by this gadget's training instead of the previous one's.

Every other line — the flushes, the 6:1 training ratio, the 16-mailbox array2 layout, the timing loop, the threshold — is unchanged, because both gadgets emit the same cache-channel shape: 16 buckets spaced 1024 bytes apart, one warmed per speculative call. The channel is gadget-agnostic; only what gets encoded into it differs.

d) [5 pts] Given the helper *pick_winner(...)* shown below, and assuming that *readMemoryByte2(...)* uses *victim_function2(...),* fill in the blank so that *leak_byte(...)* recovers a byte at any address of the attacker's choosing. Explain your answer.

```
static uint8_t pick_winner(void) {
   int best = 0;
   for (int k = 1; k < 16; k++)
       if (results[k] > results[best]) best = k;
            return (uint8_t)best;
}
uint8_t leak_byte(size_t malicious_x) {
    readMemoryByte(malicious_x);
    uint8_t a = pick_winner();
    readMemoryByte2(malicious_x);
    uint8_t b = pick_winner();
    return _____ ;    /* FILL ME and explain */
}
```

> **return (uint8_t)((b << 4) | a).** a is the low half-byte and b is the high half-byte, so shift b back into bits 4..7 and OR in a to reassemble the byte.