COTTONTAIL: Large Language Model-Driven Concolic Execution for Highly Structured Test Input Generation

Haoxin Tu
Singapore Management University
haoxintu@gmail.com

Seongmin Lee
University of California, Los Angeles
seongminlee@sigsoft.org

Yuxian Li
Singapore Management University
liyuxianjnu@gmail.com

Peng Chen
Independent Researcher
spinpx@gmail.com

Lingxiao Jiang
Singapore Management University
lxjiang@smu.edu.sg

Marcel Böhme
Max Planck Institute for Security and Privacy
marcel.boehme@acm.org

***Abstract*—How can we perform concolic execution to generate highly structured test inputs for systematically testing parsing programs? Existing concolic execution engines are significantly restricted by (1) input structure-agnostic path constraint selection, leading to the waste of testing effort or missing coverage; (2) limited constraint-solving capability, yielding many syntactically invalid test inputs; (3) reliance on manual acquisition of highly-structured seeds, resulting in non-continuous testing.**

This paper proposes COTTONTAIL, a new Large Language Model (LLM)-driven concolic execution engine, to mitigate the above limitations. A more complete program path representation, named Expressive Coverage Tree (ECT), is first constructed to help select structure-aware path constraints. Later, an LLM-driven constraint solver based on a *Solve-Complete* paradigm is designed to solve the path constraints smartly to get test inputs that are not only satisfiable to the constraints but also valid to the input syntax. Finally, a history-guided seed acquisition is employed to obtain new highly structured test inputs either before testing starts or after testing is saturated. We implemented COTTONTAIL on top of SYMCC and evaluated eight extensively tested open-source libraries across four different formats (XML, SQL, JavaScript, and JSON). The experimental results are promising: COTTONTAIL significantly outperforms baseline approaches by 30.73% and 41.32% on average in terms of line and branch coverage. Besides, COTTONTAIL found six previously unknown vulnerabilities (six CVEs assigned). We have reported these issues to developers, and four out of them have been fixed so far.

1. Introduction

Parsing software systems, such as XML and SQL libraries, are widely used in modern systems. However, even after years of intensive testing efforts, residual vulnerabilities persist, reflecting the complexity and attack surface of such components. Highly structured (or syntactically valid) test inputs are demanded to comprehensively stress the parsing test programs; as only the parser-checking logic is passed, the deeper application logic can be examined. Considerable effort has been devoted to generating structured test inputs, including black, grey, and white-box fuzzing-based approaches. Among them, white-box fuzzing via concolic execution has shown considerable capabilities of test input generation for general test programs. Given a seed input, a concolic execution engine starts by concretely executing the program while symbolically tracking the same execution path to collect path constraints. The negation of path constraints is applied to explore alternative branches. An off-the-shelf constraint solver is used to solve constraints and generate new test cases that satisfy the negated constraints, enabling the path exploration of uncovered paths. Benefiting from the soundness of test case generation and a systematic way for path exploration, it has been promising and applied in many areas [1]–[7].

Although promising, existing concolic executors (e.g., SYMCC [1] and MARCO [3]) remain significantly hindered by three fundamental limitations in their treatments for the problems of *which to solve*, *how to solve*, and *how to acquire new seed inputs* when handling parsing test programs.

#L1: The input structure-agnostic path constraints selection is either redundant or overly aggressive. When path constraints are generated during concolic execution, we need to consider the problem of *which path constraints to solve*. A straightforward idea is to select all path constraints, hoping to explore all paths in test programs. However, such a structure-agnostic option leads to many redundant path constraints, making the testing process impractical. Alternatives are to select the path constraints based on some heuristics, e.g., bit-wise coverage map `Bitmap` from SYMCC [1] and Concolic State Transition Graph (CSTG) from MARCO [3]. Unfortunately, both guides are built on the basis of binary code, which aggressively eliminates interesting code coverage. This is mainly due to the lack of expressive coverage information in the path constraints, making them difficult to distinguish and select interesting paths (detailed in §3.1.1).

#L2: The solutions from constraint solving only comply with satisfiability while neglecting syntactic validity. To

obtain high-quality (especially for highly structured) test cases by solving constraints, an important question is *how to solve the constraints*. Traditional constraint solvers (e.g., `Z3`) equipped with concolic executors usually solve constraints for satisfiability, while ignoring the syntactic validity of newly generated test cases (see more explanation in §2). Such a limitation oftentimes causes the engine to produce syntactically invalid test cases, rendering the testing effort largely in vain. We argue that an optimal solution for constraints should not only comply with satisfiability, but also be aware of syntactic validity. Also, traditional solvers can not generate new test cases with flexible sizes by design, as the number of symbolized bytes is restricted by the seed, further decreasing the effectiveness of concolic testing.

#L3: The acquisition of highly structured seeds before testing starts or after testing is saturated is difficult. Existing engines highly rely on manually collected *initial* seeds from bug repositories to start testing, where the manual work is often time-consuming. Randomly generating seeds could be an alternative, but it tends to be ineffective, as many random seeds do not boost coverage, and it would be wasteful to continue testing if the testing is saturated (i.e., code coverage has plateaued). We thus need to acquire *fresh* seeds to change the situation, but existing concolic executors are unable to generate such seed inputs during testing progress. Again, it is possible to naively feed random seeds into the testing process after saturation, but it rarely improves code coverage (as demonstrated in §5.2).

To overcome the above limitations, we propose COTTONTAIL[1], a new Large Language Model (LLM)-driven concolic execution engine to generate highly structured test inputs effectively. Our key insights are threefold. *First*, We found that for parsing programs, the inputs are processed structurally, e.g., using structured branches (either *switch-case*, *if-else*, or others) to handle different input bytes. Thus, we can build what we call *structural program paths*, i.e., paths that diverge depending on specific input values, to help not only represent meaningful and complete program paths but also reduce redundant path constraints, addressing limitation **#L1**. *Second*, given the strong input understanding and completion capabilities of LLMs [8], it could be promising to leverage them to (1) perform syntax-aware solving, i.e., *solve* the constraints for not only satisfiability but also syntax validity; (2) conduct flexible solution completing, i.e., *complete* the solution to be syntactically valid and with flexible sizes, thus alleviating limitation **#L2** (more details in §2). *Third*, benefiting from LLM knowledge and memorization [9]–[12], LLM could be induced to produce new meaningful seed inputs, thus mitigating limitation **#L3**.

Based on the above three insights, we design three new components in COTTONTAIL to explore structural program paths for highly structured test input generation.

- **Structure-aware Constraint Selection**. To address the limitation of *which to solve*, a branch information collector is first introduced during the instrumentation to help construct a more complete representation of *structural program path*. Then, based on the representation, a new coverage map, named Expressive Coverage Tree (ECT), is constructed to keep track of program branch status (e.g., taken or untaken) with expressive semantic information. Finally, guided by ECT, a path constraint selector is facilitated to conduct structure-aware path selection.
- **LLM-driven Constraint Solving**. To mitigate the limitation of *how to solve*, an LLM-driven solver, which facilitates *Solve-Complete* paradigm, is leveraged to solve the path constraints. The paradigm first *solves* constraints for satisfiability and then *completes* the solution for the syntax validity. Also, the LLM-generated test cases can have a flexible size, alleviating the restrictions of generating only test cases with fixed sizes. Moreover, to increase the robustness of LLMs, a test case validator is designed to refine unsound results from LLMs.
- **History-guided Seed Acquisition**. To address the limitation of *how to acquire new seeds*, history-guided seed acquisition strategies are designed for different timings. Before testing, we prompt LLMs to generate highly structured test inputs as *initial* seeds that may likely trigger new vulnerabilities based on their memories (e.g., from historical bug repositories). During testing, a history coverage recorder is utilized to record the mapping between the test input and its code coverage. When the coverage gets saturated, we prompt LLMs with Chain-of-Thought (CoT) [13] based on the historical mapping to generate *fresh* seeds that are likely to cover unexplored features.

In short, COTTONTAIL is a novel concolic execution engine that is capable of *structure-aware constraint selection*, *smart constraint solving*, *history-guided seed acquisition* for robust generation of highly structured test inputs. We position COTTONTAIL as a new white-box fuzzing that can effectively explore both *input space* and *execution space* to advance the field of automated testing (see more detailed comparison with existing structure-aware fuzzing in §8).

We have prototyped COTTONTAIL on top of SYMCC [1] and demonstrated its test input generation capabilities over eight widely tested libraries across four different formats (`XML`, `SQL`, `JavaScript`, and `JSON`). Our experiments show promising results. Compared with state-of-the-art approaches (i.e., SYMCC [1], MARCO [3], and their variants), COTTONTAIL significantly outperforms them by covering 30.73% more lines and 41.32% more branches on average. During the same period, COTTONTAIL significantly improves (more than 100x) the parser checking passing rate over the generated test cases. Our ablation studies also demonstrate that each component in COTTONTAIL has contributed to the better results. We have also found six previously unknown memory-related vulnerabilities and reported them to the developers (six new CVE IDs have been assigned to them, and four out of six have been fixed).

Contributions. We make the following contributions:

- To our knowledge, COTTONTAIL is the first LLM-driven concolic execution engine for highly structured test input

1. Cottontail rabbits are known for their structured running patterns (e.g., zigzagging) to evade predators using their cotton-ball tails. We used it to reflect our aim for the generation of highly structured test input.

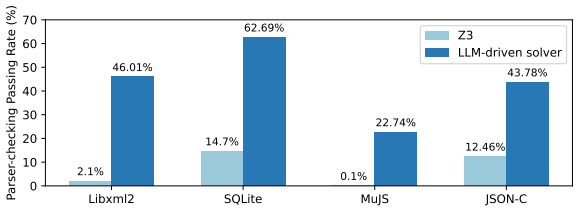

Figure 1. Comparison of parser checking pass rates between traditional solver (i.e., Z3) and LLM-driven solver (designed in COTTONTAIL).

generation, automatically working in a white-box manner.
- Three new components, including structure-aware path constraint selection, smart LLM-driven constraint solving, and history-guided seed acquisition, are designed to make COTTONTAIL effective and practical.
- Extensive experiments are conducted to demonstrate the capabilities of COTTONTAIL. The results show that COTTONTAIL can not only significantly outperform baseline approaches but also is practical to find new vulnerabilities.
- The prototype COTTONTAIL is open source[2] to foster future research that combines program analysis and LLMs.

2. Background and Motivation

Concolic Execution. Concolic execution, also known as dynamic symbolic execution, integrates symbolic and concrete execution to explore program paths systematically. Concrete values guide the actual execution path, ensuring feasibility, while symbolic values enable exploration of alternative paths by generating new test cases. In recent years, compilation-based concolic execution (e.g., SYMCC [1] and MARCO [3]) has gained popularity due to its superior performance and practical applicability. In particular, concolic execution is a key component of the tools for the winning team for both old DARPA CGC [14], [15] and recent AIxCC Atlanta [16]. Technically, given an initial seed input, these engines embed symbolic reasoning/tracing logic directly into compiled binaries and collect path constraints at runtime. By negating selected path constraints, the engine produces new constraints representing unexplored branches, which are then solved using off-the-shelf constraint solvers (e.g., Z3 [17]) to generate new test cases. Note that the new test cases usually hold the same size as the seed input because the size of symbolic bytes is fixed when the seed input is fed into the concolic engine in the initial phase. The concolic testing process is continued by iteratively feeding the new test cases back into the execution loop, which is ideally an endless process. However, concolic testing naturally encounters a saturation point when it can no longer cover new code due to the lack of diversity in test inputs (e.g., limited by the input size) [15], [18], [19] or the restricted covering capability of harnesses/test drivers [20]–[22]. When testing is saturated, the best practice is to acquire *fresh* new seeds that drive the exploration forward for long-term continuous testing. In short, *which to solve* determines both effectiveness and efficiency of testing, *how to solve*

2. https://github.com/Cottontail-Proj/cottontail

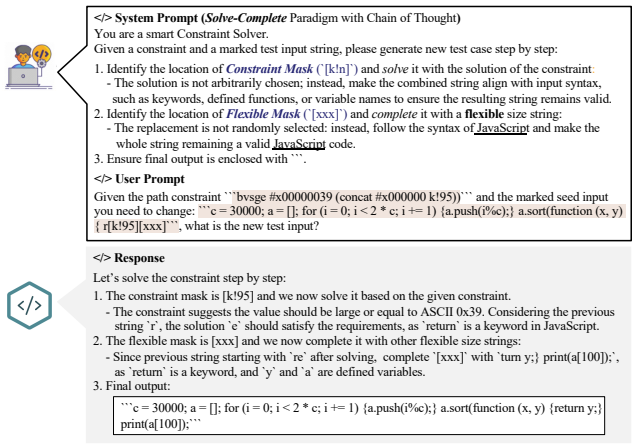

Figure 2. The *Solve-Complete* paradigm and LLM's response. In the upper box, the underlined text represents different formats, and the colored text enclosed indicates either constraints or a marked test input string.

affects the effectiveness of test case generation, while *how to acquire new seeds* decides the continuity of testing.

LLMs for Test Input Generation. Large Language Models (LLMs) are popular AI systems designed to predict the next word or token in a sequence based on the context of preceding tokens. Recent emergence of LLMs has driven their application in numerous security-related domains [23]–[25]. However, most of the LLM-based systems lack analytical depth and robustness, which limits their effectiveness in more complex scenarios such as systematic program understandings [26]. Two promising directions to mitigate the problem are either to integrate advanced program analysis or design logical Chain-of-Thought (CoT) prompts to further improve the reliability and robustness of the LLM-based system [27], [28]. While LLMs have been combined with black-box [29], [30] or grey-box [20], [31] fuzzing techniques for structured test input generation, to our knowledge, no study has yet attempted to integrate LLMs with more systematic techniques like concolic execution to enhance the security analysis. We believe it could be promising to empower the potential of LLMs for more rigorous security guarantees by integrating LLMs with concolic testing.

Motivation: Investigative Study. It is evident that there is a need for a new path constraint selection strategy to select optimal subsets of constraints and a new seed acquisition to generate highly structured inputs to improve both effectiveness and continuity of concolic testing. It may not be clear why we need an LLM-driven solver (which served as our core innovation) to generate highly structured test inputs.

To justify our motivation, we conducted an investigation study to show the significant limitations of traditional constraint solvers and how an LLM-driven constraint solver can mitigate them. To do so, we first selected four libraries that process four different input formats (XML/SQL/JavaScript/JSON) as test programs, and ran them using the existing concolic executor SYMCC. Then, we collected the generated path constraints and used a traditional solver Z3 to solve them to obtain new test

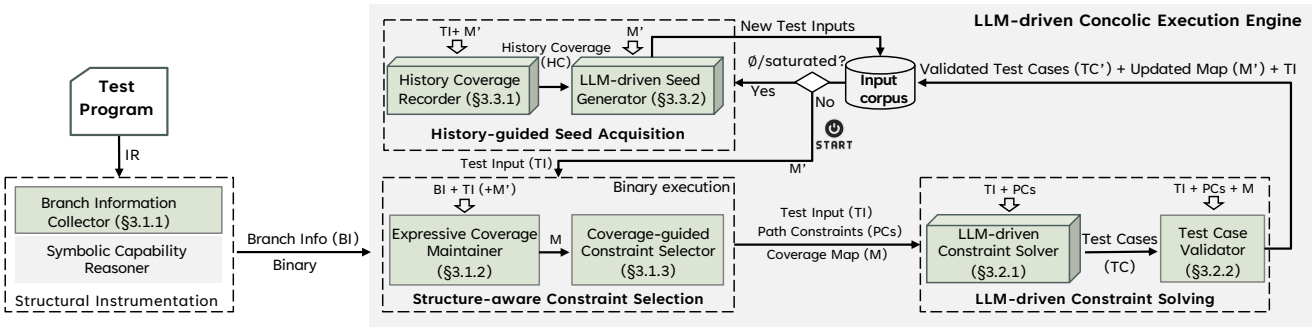

Figure 3. High-level design of COTTONTAIL concolic execution engine

cases. After running the new test cases with four target programs, we found that numerous test cases (85.3% to 99.9%, as shown in Figure 1) are syntactically invalid and cannot pass the parser checking logic. This is reasonable as existing solvers only solve constraints for satisfiability while ignoring the guarantee of syntactic validity of the solution by design. Also, the size of the resulting solution is fixed, restricting the diversity of generated test cases. For example, given a path constraint "*bvsge #x00000039 (concat #x000000 k!95)*" (where 'k!95' is a symbolic variable, collected from parsing implementation in Figure 4) which requires the value of the symbol 'k!95' to be large or equal to ASCII 0x39 (i.e., char '*9*'). `Z3` simply solves it to '*9*' and keeps the remaining string unchanged with fixed size as seed input, producing an invalid JavaScript string '*r9turn* ...'. This fact indicates that the test cases generated by existing concolic executors can hardly examine the deeper code regions (e.g., the application logic), significantly hindering the effectiveness of concolic testing. In summary, such a limitation motivates us to investigate the following question: *How to solve the constraints smartly, i.e., solving constraints for both satisfiability and syntactic validity as well as making the resulting solutions with flexible sizes*?

To answer the question, we design a new LLM-driven constraint solver based on *Solve-Complete* paradigm as shown in Figure 2. The idea behind it is simple but effective. Given a path constraint and the marked (two types of marks help *Solve-Complete* paradigm) seed input string as a user prompt, LLMs are prompted to smartly solve the given constraint within two consecutive steps. In step (1), LLMs are asked to *solve* the constraint and use the solution to replace the *Constraint Mask* '[k!n]', where the solution is syntax-aware, that is, supposed not only to meet the satisfiability but also possibly comply with the input syntax validity. In step (2), LLMs are required to *complete* the *Flexible Mask* '[xxx]' with a flexible size string which makes the whole resulting string remain valid. Taking the same path constraint includes 'k!95' again, our solver solves it to '*e*', which can be connected with a remaining string (e.g., '*turn*') to form a syntactically valid JavaScript string (as shown in the LLM's response). To demonstrate the effectiveness of the LLM-driven solver, we used it to solve the same path constraints solved by `Z3`, and the results presented in Figure 1 show significant improvements (100x more) in terms of the parser checking pass rate across various formats, practically helping achieve significant improvement in terms of code coverage.

3. Design of COTTONTAIL

Overview. Figure 3 presents the high-level design of COTTONTAIL. Given a test program, COTTONTAIL enhances instrumentation by collecting branch information at compile-time and constructs a new program path representation called Expressive Coverage Tree (ECT) at runtime to assist in structure-aware constraint selection (§3.1). Later, COTTONTAIL leverages LLM-driven constraint solving with *Solve-Complete* paradigm to solve constraints smartly and refine the non-robust results from LLMs (§3.2). Finally, if there are no initial seed inputs to set up testing or whenever the testing process gets saturated, COTTONTAIL adopts a history-guided seed acquisition to obtain high-quality seed inputs continuously (§3.3).

3.1. Structure-aware Constraint Selection

This subsection explains why a new path constraint selection strategy is needed and details how we construct a new coverage map to guide the structure-aware selection.

3.1.1. Why Structure-aware Selection. To perform a practical concolic execution over parsing test programs, we argue that an effective selection should be structure-aware, which:

- #1 provides a meaningful (e.g., includes semantic information) and complete representation of program paths;
- #2 records human-readable coverage information;
- #3 excludes redundant structure-agnostic path constraints;
- #4 has less chance of missing interesting coverage.

Satisfying requirement #1 helps users to have a better understanding of structural paths, #2 is essential to help craft useful inputs either by humans or LLMs at runtime when the testing gets saturated. Compared with branch information from binary code (which contains less semantic information), a human-readable coverage could provide clear and interpretable insights into which branches have

```
1  // Parsing logic /* jslex.c */
2  static int jsY_isidentifierpart(int c) {
3     return isdigit(c) //"bvsge #x00000039 (concat #x000000 k!95)"
4     || isalpha(c) || c == '$' ||c == '_' ||isalpharune(c);
5  }
6  static int jsY_lexx(js_State *J){
7     while (1) {
8        // ...
9        switch (J->lexchar) {
10          case '(': jsY_next(J); return '(';
11          case ')': jsY_next(J); return ')';
12          case ',': jsY_next(J); return ',';
13           // ...
14       }
15    // ...
16    }
17 }
```

Figure 4. Sample parsing implementation code from MuJS

been covered and what direction to create new test cases. #3 and #4 together guarantee a better trade-off between testing effectiveness and efficiency. There are three existing strategies to handle the path constraint selection for the general software systems, which are exhaustive search, the Bitmap-based approach [1], and the CSTG-based approach [3], yet they do not comply with all the requirements when handling parsing test programs. Justified by the above facts, it thus calls for a new path constraint selection that could satisfy all four requirements. In the following subsections, we detail the structural instrumentation and human-readable Expressive Coverage Tree (ECT) to meet requirements #1 and #2, and the ECT-guided path constraint selector to comply with requirements #3 and #4.

3.1.2. Structural Instrumentation. Compilation-based concolic execution has shown promising performance in execution speed compared with IR (Intermediate Representation)-based execution, but it inevitably misses many interesting behaviors of the test programs due to the loss of semantics information after compilation [1], [2]. To alleviate this issue, we propose to use extra instrumentation on the IR code to collect necessary branch information. It is worth noting that such instrumentation is crucial for capturing meaningful structural information, particularly around complex branch conditional constraints. This is because it allows developers and analysis tools to observe precisely which paths of the code are being exercised at runtime. Without such instrumentation, it can be difficult to determine whether certain cases or branches within a switch statement are triggered, leading to potential blind spots in testing. To comprehensively cover structural program paths, we design a branch information collector to capture all possible structural paths during instrumentation. To be specific, our structural instrumentation systematically walks through every instruction in a given function, looking for branch instructions such as *switch*. When it encounters a branch, it records the branch name and its associated case values. Such metadata is then added to a global map that associates each branch statement with its case values. Finally, the instrumentation phase saves all gathered information into a `JSON` file, enabling further analysis of the function's branching structures and program semantics.

3.1.3. Expressive Coverage Tree Maintainer. After collecting branch information, we introduce a new Expressive Coverage Tree (ECT) to help have a comprehensive representation of structural program paths.

We define ECT as a hierarchical tree structure represented by a pair $T = (N, E)$, where:

- N is a set of nodes, containing a special root node.
- $E \subseteq N \times N$ is a set of edges that define parent-child relationships, representing the call context (caller to callee) or branch information (condition to nested statement).

Each node $n \in N$ may have zero or multiple child nodes connected by edges. Nodes without children are called *leaf* nodes; nodes with children are referred to as *internal* nodes. Each node name is a unique identifier:

fileName_funcName_lineNum_colNum_brType_brId

which consists of several important attributes to represent a unique branch or differentiate between different branches. Those attributes include visiting status (*taken* or *untaken*), visit count (*visit_cnt*), branch type (*brType*), call stack size, and branch id (*brId*) — in *if* statement, it refers to 0 (*then* branch) or 1 (*else* branch); in *switch-case* statement, it represents the constant case value. With the help of the expressive coverage map, users can easily understand the testing process by checking the statistics in the global map.

```
{
"loc": "jslex.c_jsY_lexx_9_3_switch", "tp": 1, "tk": 1,
    "cs": 10, "vc": 1, "br": -1,
  "ch": [
  { "loc": "jslex.c_jsY_lexx_9_3_switch_40",
      "tp": 1, "tk": 1, "cs": 10, "vc": 1, "br": 40 },
  { "loc": "jslex.c_jsY_lexx_9_3_switch_41",
      "tp": 1, "tk": 1, "cs": 10, "vc": 1, "br": 41 },
  { "loc": "jslex.c_jsY_lexx_9_3_switch_44",
      "tp": 1, "tk": 1, "cs": 10, "vc": 1, "br": 44 }
  ]
}
```

Figure 5. Partial expressive coverage tree (ECT) recording the program paths between Lines 9-12 in Figure 4 in JSON format (*loc*: source code location; *ch*: children; *tp*: branch type (0 for *if* statement and 1 for *switch* statement); *tk*: taken; *cs*: call stack size; *vc*: visited count; *br*: branch id).

To provide a clearer understanding of the ECT used in COTTONTAIL, Figure 5 presents a partial ECT that captures the coverage information of the switch statement with three children in Figure 4 at Line 9. Once the function `jsY_lexx` is analyzed, the ECT of the program in JSON format is recorded in the global coverage map. It is worth noting that the ECT differs from existing code coverage trees (e.g., CSTG from Marco [3]) or maps (e.g., the bitmap from SymCC [1]): It is *partially* context-sensitive and path sensitive, capturing not only branch coverage information but also call stack information. Furthermore, since our representation is based on source code instead of binary code, we can avoid the loss of interesting path coverage or semantic information (e.g., detailed *switch-case* branches to store structured information) due to hash collision or source code compilation. In short, we provide a precise representation of code coverage that enables the distinction of different execution contexts for the same branch, thereby

```
1 (vsge #x00000039 (concat #x000000 k!0)
2 (vsge #x00000039 (concat #x000000 k!1))
3 ....
4 (vsge #x00000039 (concat #x000000 k!95))
```

Figure 6. Duplicated path constraints in parsing logic (those path constraints aim to cover the same branch at Line 3 in Figure 4)

facilitating systematic exploration of the input space and advancing automated test generation.

After defining ECT, we manage and update a global coverage tree to help guide the constraint selection. Thus, COTTONTAIL is able to reduce redundant coverage but avoid losing promising code coverage.

3.1.4. ECT-guided Constraint Selector. The selector consists of two phases of reduction: reducing redundant constraints that do not increase coverage during single concolic execution, and reducing constraints across different runs that have less chance to boost code coverage.

First Phase Reduction. As aforementioned in Figure 4, in a single concolic execution, it is common that the redundant path constraints are collected when each input byte is repeatedly analyzed by a parsing function. Therefore, we need to remove such duplicated path constraints. To do so, we maintain a global branch recorder to record branch-constraint mappings during a single concolic execution. Since each element in the record is a unique branch identifier that records the context of the branch. When a branch is encountered and already exists in the recorder, we compare the current constraints and the constraints stored in the branch. If the constraints are only different in the symbolic index (i.e., each byte represented as 'k!n' in path constraints, where 'n' indicates the index over the input bytes), as shown in Figure 6, we reduce a duplicate set of path constraints and only keep unique branches.

Note that such a constraint deduplication mechanism preserves the soundness of concolic execution by excluding only redundant path constraints that arise from structurally identical branches applied repeatedly across input positions (also demonstrated in first phase selection in Table 3).

Second Phase Reduction. Different from the first phase, in the second phase, we remove redundant path constraints based on the newly built coverage tree across different runs. An important problem to handle is how we remove path constraints without affecting the overall performance (i.e., missing potential interesting code coverage). Simply excluding the branches that have been explored tends to miss much interesting coverage, as such a strategy does not have a chance to examine the remaining execution of the current test case or the remaining test cases to make a globally optimal decision [3]. Therefore, we need to be careful when making the selection decision. Inspired by prior work, such as KLEE [32] and Mayhem [15], [33], we consider factors including untaken branches, frequency of visits, and depth of execution to balance exploration potential and redundancy, as each coefficient has been shown to contribute to uncovering previously unexplored paths.

Algorithm 1: Test Case Validator

Input: a path constraint *pc*, a branch *br*, a test input from LLM *input*, the coverage tree *g_tree*
Output: original test input *intput* or refined test input *input'*, updated global tree *g_tree'*

```
1  Function TestCaseVal(pc, br, input, g_tree):
2      res_eva = evalauteConstraint(pc, input)
3      if res_eva == True then
4          updateGlobalTree(g_tree, br)
5          return input, g_tree'
6      else
7          solution = getSolution (pc)
8          input' = refineTestCase (solution, input)
9          updateGlobalTree(g_tree, br)
10         return input', g_tree'
```

Thus, we propose a new metric node weight to quantify selection priority, defined as follows:

$$Node_{weight} = \alpha \cdot untaken + \beta \cdot visit\_cnt + \gamma \cdot depth \quad (1)$$

The *untaken*, *visit_cnt*, and depth are node attributes, and α, β, and γ are three parameters to optimize exploration for maximum program coverage. α prioritizes untaken branches, ensuring the discovery of new execution paths and highlighting untested code regions. β focuses on rarely visited nodes, balancing exploration by avoiding overemphasis on frequently traversed paths while paying attention to less common scenarios. γ rewards deeper nodes, encouraging exploration of complex execution paths and uncovering deeply nested bugs or vulnerabilities. Together, these parameters enable a balanced trade-off that drives efficient and thorough program testing.

3.2. Smart LLM-driven Constraint Solving

This subsection introduces our *Solve-Compete* paradigm to smartly solve path constraints and a new test case validator to refine the unreliable results produced by LLMs.

3.2.1. LLM-driven Constraint Solver. It is important to give a precise and logical prompt if we intend to receive output feedback from LLMs (we show that normal prompts generate worse results in §5.2). We design a *Solve-Complete* paradigm that utilizes the CoT prompt mechanism. Using CoT prompts instead of normal prompts is advantageous for tasks requiring complex reasoning or multistep problem-solving. CoT prompts guide the model to think step-by-step, improving accuracy by reducing errors that arise from skipping intermediate steps. It also enhances transparency by explicitly laying out the reasoning process, making it easier to verify the logic and correctness of the solution. In summary, the systematic reasoning makes CoT prompts ideal for tackling constraint-solving tasks.

The earlier Figure 2 illustrates a smart constraint-solving strategy grounded in a *Solve-Complete* paradigm, where the LLM is asked first to satisfy path constraints and then

complete the output to preserve syntactic correctness. This process is decomposed into two stages: (1) resolving the *Constraint Mask* ('[k!n]') (e.g., 'k!95') using a syntax-aware way by synthesizing a character 'e' that satisfies the path constraint under ASCII semantics, and (2) completing the surrounding code such that the entire string remains valid JavaScript to fill the *Flexible Mask* ('[xxx]') with a flexible size. This dual-stage approach mirrors classical symbolic execution techniques, but is uniquely enhanced by the LLM's ability to generate structurally and contextually coherent test inputs. In contrast to traditional program synthesis pipelines, which often treat constraint solving and code completion as decoupled steps, this strategy tightly integrates reasoning with generative synthesis. The mechanism also aligns with tasks like code infilling, notably benchmarked in CodeXGLUE [8], where models are expected to fill in masked code spans while preserving functional correctness. However, unlike pure statistical infilling, our approach exhibits explicit constraint awareness, solving constraints before code generation, highlighting the potential of LLMs to unify symbolic reasoning with syntax-preserving code completion. In summary, the systematic reasoning capability of the *Solve-Compete* paradigm unlocks new applications in symbolic execution for structured test input generation.

3.2.2. Test Case Validator. There is a known issue that LLMs can not reliably generate expected output and can have hallucinations [27], [34]. Random output might be acceptable for black/grey box fuzzing, as they do not require the robust (i.e., new test inputs will cover new code coverage) results during each iteration. However, for a concolic execution, robustness is one of the essential features that should be guaranteed. Therefore, we need to handle unreliable results produced from GPT to ensure it follows the soundness guarantee of traditional concolic execution and updates the global ECT precisely.

Algorithm 1 presents the workflow of the test case validator, which validates or refines test inputs generated by LLMs. The algorithm takes as input a path constraint *pc*, a branch *br*, the test input (*input*) produced by LLMs, and the global coverage tree (*g_tree*). It first evaluates whether the input satisfies the path constraint using the function `evaluateConstraint` (Line 2). If the constraint is satisfied, i.e., the returned boolean flag *res_eva* is *True*, the branch *br* is updated in the global tree using `updateGlobalTree` (Line 9), and the algorithm outputs the original test input (*input*) alongside the updated tree. If the constraint is not satisfied, a solution for the path constraint is computed using `getSolution` (Line 7), and a refined test input (*input'*) is generated through function `refineTestCase` (Line 8). Finally, the updated tree (*g_tree'*) is returned along with the refined input. In summary, this algorithm ensures the validity of test cases and updates the global coverage to improve test coverage.

In particular, in `refineTestCase` function, we replace the unreliable solution generated by LLM with the correct solution generated by a traditional solver. By such, even though LLMs produce unreliable outputs, COTTONTAIL could fix them and refine them to the same output as the ones generated by traditional solvers.

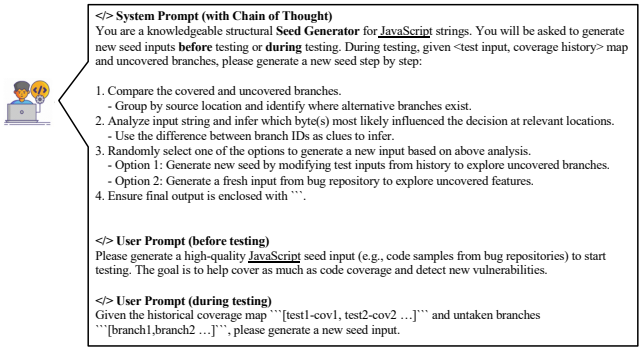

Figure 7. Chain of Thought (CoT) prompts for LLM-driven seed generation

3.3. History-guided Seed Acquisition

In this subsection, we detail the strategy to generate initial seeds before testing or alleviate the saturation issue during testing, including the history coverage recorder and the history-driven seed generator. Since the key contributions lie in the generation during testing, we detail the history coverage recorder first in the following.

3.3.1. History Coverage Recorder. It is important to trace the testing history to know *which branches* can be covered by *what test inputs* and *which are the uncovered branches* remaining uncovered. By investigating the connection between test input and its covered branches, we could not only understand the underlying processing logic of test programs but also highlight what missing features are within the test inputs. To practically collect the history information, we continue to leverage the benefits of the informative coverage map (i.e., ECT) to get the covered or uncovered branches.

After collecting history coverage mappings and extracting branch information from the global coverage map, we then use this information to construct CoT prompts for *fresh* seed generation during testing.

3.3.2. LLM-driven Seed Generator. The generator is invoked based on two different timings: initial seed acquisition before testing and fresh seed acquisition during testing.

Initial Seed Acquisition. If there are no interesting seed inputs to set up testing, we define a prompt that helps generate high-quality structured seed inputs. Since LLMs were trained via tons of code and resources and inspired by many existing studies [29], [30], it is reasonable that LLMs have expert knowledge of what kinds of code have triggered vulnerabilities in bug repositories. Thus, we directly prompt LLMs to generate high-quality structured test inputs from existing bug repositories. By such, no manual work will be required to collect historical buggy code examples.

Fresh Seed Acquisition. During testing, if there is no interesting coverage increase (i.e., saturated) after a timeout

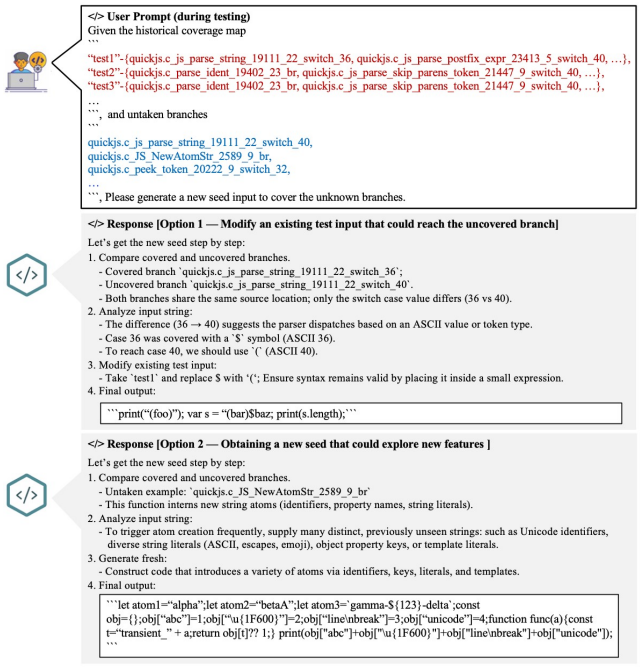

Figure 8. Examples of LLM responses of history-guided seed generation

(i.e., three minutes), by check the coverage (collected from external tool `gcov`) at runtime. It is straightforward to apply the same strategy used in seed generation to get a fresh seed, but it is ineffective (demonstrated in §5.2). To make it more effective, we design a creative generation solution using CoT (Chain of Thought) prompts to effectively explore the unexplored branches/features during testing. Such a design is inspired by an interesting behavior investigated by prior studies, i.e., a better name can help LLMs better understand the program semantics [12], [35].

Figure 7 illustrates the prompts designed to guide a seed generator for `JavaScript` in creative generation of seed inputs for different timings. In particular, the CoT workflow guides an LLM to generate new `JavaScript` seed input through a structured multi-step reasoning process during testing. First, the LLM compares covered and uncovered branches, groups them by source location, and identifies divergent execution points. Then, it analyzes input bytes likely responsible for branching decisions using its internal inferring capacities. Finally, it synthesizes new inputs either by mutating existing test cases to explore specific branches or by drawing from existing bug repositories.

Examples of LLM-driven Seed Generation. To have a better understanding of how history-guided seed acquisition works, we provide two illustrative examples in Figure 8 to articulate the seed generation process (for simplicity, only a reduced version is presented). In principle, COTTONTAIL learns from historical coverage and generates new seeds in two ways: 1) mutating historical test inputs; 2) generating new seeds (or test inputs) from scratch.

In the first response shown in Figure 8, COTTONTAIL generates a new test input by mutating a historical test input that has been executed before. The historical test input is shown in the prompt, and COTTONTAIL asks the LLM to generate a new test input by making some changes to it. The generated test input is similar to the original but contains differences that may lead to the exploration of new program paths. This strategy is effective because mutating existing test inputs leverages prior knowledge to produce new inputs that are both likely to be valid and potentially interesting.

In the second response, COTTONTAIL generates a new test input from scratch by leveraging the naming conventions used in the target program's implementation. This choice is motivated by the expectation that good programmers typically employ meaningful and consistent naming patterns for the intentions of functions [36]–[38]. Moreover, recent studies show that LLMs can utilize such identifier information to assist code analysis [12], [39].

Note that the generated seeds may not always satisfy the requirements for covering the untaken branches. Instead, we expect them to provide useful hints that guide the exploration of the input space, thereby increasing the likelihood of generating a test input that can trigger those branches. If a new seed fails to cover the target branches, COTTONTAIL will quickly discard it guided by the ECT.

4. Implementation

We implemented COTTONTAIL on top of SYMCC (commit version 65a3633). The newly designed components structural instrumentation and coverage-guided path constraints selection (§3.1) are implemented as separated functions using C++. The remaining LLM-driven constraint solving (§3.2) and history-guided seed acquisition (§3.3) are implemented in Python code. The running script is set up using Python 3.9. For the setting of different parameters α, β, and γ used in Equation 1, we run extra experiments and opt for one setting that has a better trade-off between efficiency and effectiveness, which is 1.0, 3.0, and 0.8, respectively. We used `gpt-4o` (only in Setting 1 in RQ1) and `gpt-4o-mini` as our base LLMs and invoked their Python APIs to communicate with the model. The temperature of the model is set to 0 for better reproducibility. Note that an LLM with better reasoning capabilities (or higher cost) is preferred but not required. Our extra experiments show that other recently released cost-effective models (i.e., `deepseek-v3` and `gpt-4.1-nano`) can work very well compared with higher cost models such as `gpt-4o`.

5. Evaluation

To evaluate the effectiveness of COTTONTAIL, we aim to investigate the following research questions (RQs):

- **RQ1**: How does COTTONTAIL perform compared with state-of-the-art concolic execution approaches?
- **RQ2**: Can each component contribute to COTTONTAIL?
- **RQ3**: Can COTTONTAIL find new vulnerabilities?

Among these RQs, RQ1 focuses on demonstrating the effectiveness of COTTONTAIL compared with state-of-the-art approaches and investigating whether COTTONTAIL is

TABLE 1. OPEN SOURCE LIBRARIES CROSS FOUR DIFFERENT FORMATS USED IN EVALUATION (LOC: LINES OF CODE; STARS: GITHUB STARS)

Libraries	Format	Version	LOC	Stars
Libxml2	XML	2.13.5	80.0k	0.6k
Libexpat	XML	2.6.4	14.6k	1.1k
SQLite	SQL	3.47.0	81.3k	7.0k
UnQLite	SQL	1.1.9	22.5k	2.1k
MuJS	JavaScript	1.3.5	10.0k	0.8k
QuickJS	JavaScript	0.7.0	46.4k	1.2k
JSON-C	JSON	0.18	4.7k	3.0k
Jansson	JSON	2.14	5.8k	3.1k

superior to them. RQ2 conducts comprehensive ablation studies to analyze the significance of individual components or key features within COTTONTAIL. RQ3 assesses the practical vulnerability detection capability of COTTONTAIL.

All experiments were run on a Linux PC with Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz x 12 processors and 64GB RAM running Ubuntu 18.04 operating system.

Benchmarks. Table 1 presents eight widely tested open-source libraries used for evaluation, including libraries for `XML` (Libxml2/Libexpat), `SQL` (SQLite/UnQLite), `JavaScript` (MuJS/QuickJS), and `JSON` (JSON-C/Jansson), varying in size and popularity. This diverse set of libraries covers a broad range of formats, codebases, and community adoption levels, making it a comprehensive benchmark suite for evaluation.

5.1. RQ1: Comparison with Baseline Approaches

Comparative Approaches. The following state-of-the-art concolic execution approaches are compared:

- SYMCC [1]: the tool COTTONTAIL built on top of (enable the `Bitmap`-guided path constraints selection by default).
- SYMCC(¬MAP): a variant approach of SYMCC that selects all newly generated path constraints.
- MARCO [3]: a recent concolic execution engine that constructs CTSG to select path constraints.
- MARCO(MC): A variant of MARCO that adopts Markov Chain modeling in CSTG.
- MARCO(CFG): A variant of MARCO that applies the CFG-directed searching algorithm in CSTG.

We select SYMCC and SYMCC(¬MAP) as we built COTTONTAIL on SYMCC. MARCO is an approach proposed recently, and its experiments show that the two variant approaches (i.e., MARCO(CFG) and MARCO(MC)) could outperform MARCO in a few cases, so we also include them.

Running Settings. We design three distinct settings to comprehensively demonstrate the superiority of COTTONTAIL.

- Setting 1: We run each approach with a timeout of 1 hour.
- Setting 2: We run COTTONTAIL with a timeout of 1 hour and other approaches with a timeout of 12 hours.
- Setting 3: We set a 12-hour timeout for each approach.

We selected a timeout of 1 hour in Settings 1 and 2 for two reasons. First, we found that COTTONTAIL not only

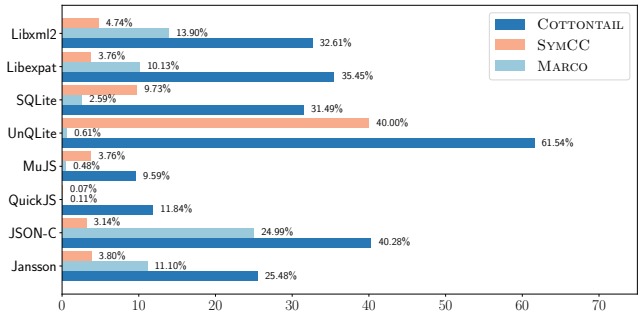

Figure 9. Comparison results of parser checking passing rate (% in *y-axis*) against SYMCC and MARCO

significantly improves the baseline approaches within one hour, but *the results of running our approach for only 1 hour can even beat the baselines running for 12 hours on average*. Second, as suggested by prior studies [40], [41] and our experimental results in Setting 3, while having longer testing campaigns can boost the effectiveness, we found that the gain when increasing the time budget after 1 hour is overall relatively poor. We run with a longer running time in Setting 3, as this could help us understand when the testing gets saturated and justify the need for a new seed generation strategy. COTTONTAIL outperforms baselines in all three settings on average, but due to the page limit, we only report detailed results of Setting 1 in RQ1 and leave other results in the Appendix.

To further conduct a fair comparison, we use the seeds from MARCO and launch each tool with the same seeds for all settings. To help detect possible program issues, we compile the target program built with `AddressSan` [42] and use it as a test oracle to detect memory issues. In particular, although COTTONTAIL does not require pre-collected seed inputs to set up, for a fair comparison, we disable the seed acquisition contribution in COTTONTAIL. To clarify, the description of COTTONTAIL in this subsection refers to COTTONTAIL(INIT+¬SGEN), the variant version of COTTONTAIL where the same initial seeds as the baselines are used and without new seed generation (please check different versions of COTTONTAIL in §5.2). To reduce the threats from randomness, we repeated running them five times and reported the median results.

Metrics. We use code coverage, including line and branch coverage measured by the external tool `gcovr` to compare the effectiveness of comparative approaches.

Results. Table 2 provides a comprehensive comparison results achieved by comparative approaches. Notably, COTTONTAIL significantly achieves a superior code coverage on average, from 20.77% to 38.16% (30.73% on average and up to 99.65% improvement over QuickJS) in terms of line coverage and 25.56% to 57.70% (41.32% on average and up to 175.88% over Libexpat) in terms of branch coverage against comparative approaches, yielding up to 12k more lines and 9k more branches in total. Note that the very recent approach MARCO can only achieve 13% code coverage than

TABLE 2. LINE AND BRANCH COVERAGE COMPARISON RESULTS AGAINST EXISTING CONCOLIC EXECUTION ENGINES SYMCC [1] AND MARCO [3]

Fomat	Libraries	SYMCC		SYMCC(¬MAP)		MARCO		MARCO(MC)		MARCO(CFG)		COTTONTAIL	
		Line	Branch	Line	Branch	Line	Branch	Line	Branch	Line	Branch	Line	Branch
XML	Libxml2	3,917	2,693	3,917	2,667	5,252	3,790	4,197	3,027	4,170	3,013	**5,298**	**3,848**
	Libexpat	2,379	1,250	2,430	1,465	1,902	1,271	1,861	1,165	1,569	680	**2,838**	**1,876**
SQL	SQLite	17,143	10,827	16,157	10,048	11,768	7,229	11,696	7,148	12,105	7,383	**18,708**	**11,849**
	UnQLite	2,927	1,503	3,123	1,619	3,087	1,629	3,062	1,677	3,230	1,720	**3,257**	**1,749**
JavaScript	MuJS	3,669	1,815	3,573	1,703	2,947	1,588	2,857	1,464	2,634	1,184	**4,120**	**2,070**
	QuickJS	5,890	2,389	4,971	1,829	6,756	2,912	6,337	2,686	5,748	2,251	**9,414**	**4,266**
JSON	JSON-C	968	568	904	488	990	637	867	527	886	547	**1,040**	**686**
	Jansson	1,021	578	571	295	1,084	650	1,089	657	1,092	659	**1,140**	**699**
	Amount	37,914	21,623	35,646	20,114	33,886	19,706	31,966	18,351	31,434	17,437	**45,815**	**27,043**
	Impr. (num)	+7,901	+5,420	+10,169	+6,929	+12,029	+7,337	+13,849	+8,692	+14,381	+9,606	-	-
	Impr. (%)	20.77%	31.64%	36.44%	53.83%	25.44%	26.56%	32.81%	36.87%	38.16%	57.70%	-	-

baseline approaches. Furthermore, in both Settings 2 and Setting 3, COTTONTAIL consistently outperforms all comparative approaches on average, demonstrating the stronger capabilities on code coverage.

To have a better understanding of why COTTONTAIL is superior, we further analyze the validity of generated test cases among SYMCC, MARCO, and COTTONTAIL. Since SYMCC and MARCO produced millions of test cases in 12 hours, we ran them in another 1-hour setting for the same running time. Figure 9 presents a detailed comparative analysis of parser checking passing rates for three comparative tools. From the figure, we can observe that COTTONTAIL consistently performs better in several critical libraries: it achieves a significant 32.61% passing rate in Libxml2, yielding 588% higher rate than SYMCC's 4.74% and 140% higher rate than MARCO's 13.90%. The overall results suggest that the increased number of valid test inputs helps yield better code coverage results.

Answer to RQ1: COTTONTAIL significantly improves the state-of-the-art approaches in terms of line/branch coverage on average in all three running settings, demonstrating the effectiveness of COTTONTAIL in generating highly structured test inputs.

5.2. RQ2: Ablation Studies

This subsection presents the methodologies to evaluate the impact of the newly designed components.

RQ2.1: How effective is the ECT-guided path constraint selection? As mentioned in §3.1.4, we design two phases to remove redundant path constraints across single concolic execution and in-between runs. We here evaluate how many path constraints were filtered out in the two phases (e.g., single or in-between concolic execution), comparing with existing selection strategies (i.e., select all without map `Nomap` and guided by `Bitmap`).

To do so, we run the seed input and terminate it after the first iteration is done to evaluate the effectiveness in the first phase. Then, we run the seed input within two iterations to evaluate the effectiveness of the in-between runs. Finally, we count the total number of path constraints and code coverage achieved by different selection strategies in the two phases.

The results in Table 3 demonstrate the effectiveness of ECT in guiding path constraint selection across both testing phases. In the first phase, ECT significantly reduces the number of collected path constraints compared to both `NoMap` and `Bitmap` (e.g., 67 vs. 223 and 106 in JSON-C; 137 vs. 401 and 191 in SQLite), while maintaining identical line coverage, indicating that ECT effectively filters redundant constraints without sacrificing exploration. In the second phase, ECT continues to show a substantial reduction in the number of path constraints relative to `NoMap` (e.g., 926 vs. 7,024 in JSON-C; 687 vs. 3,618 in SQLite), yet it retains more path constraints than `Bitmap`, enabling it to achieve better coverage than `Bitmap` and comparable or even slightly improved coverage over `NoMap` in most benchmarks. These results highlight ECT's superiority in balancing structure-aware constraint selection. The superiority is reasonable as AFL's `Bitmap` tends to miss interesting coverage due to hash collisions, limited granularity, and lack of path sensitivity, all of which cause distinct behaviors to appear identical, reducing fuzzing effectiveness [43], [44], while using `Nomap` will lead to inefficient testing. Note that the size of ECT depends on the complexity of the test programs. For example, the size is about 58.3KB for `JSON-C` after 12 hours of running.

RQ2.2: How effective are the CoT prompts in** Solve-Complete **paradigm? We have shown the superior performance of LLM-driven constraint solving in Figure 1. To better understand the benefits of the CoT prompt, we compare COTTONTAIL with COTTONTAIL(NORMPRO), a variant of COTTONTAIL that removes the CoT prompt.

The results in Figure 10 highlight the effectiveness of Chain-of-Thought (CoT) prompts in improving constraint solving for line coverage across a diverse set of libraries. In all cases, COTTONTAIL with CoT prompts (dark bars) achieves higher coverage than the variant using normal prompts (light bars), with particularly notable improvements observed in SQLite, QuickJS, and MuJS. The substantial gain in SQLite, where coverage increases from approximately 13,000 to over 15,000 lines, underscores how step-

TABLE 3. RESULTS OF PATH CONSTRAINT SELECTOR DESIGN IN COTTONTAIL

Libraries	Comparison of Path Constraints Selection — First Phase						Comparison of Path Constraints Selection — Second Phase					
	NoMap		Bitmap		ECT (ours)		NoMap		Bitmap		ECT (ours)	
	No.pc	*Cover.*	*No.pc*	*Cover.*	*No.pc*	*Cover.*	*No.pc*	*Cover.*	*No.pc*	*Cover.*	*No.pc*	*Cover.*
Libexpat	461	1,441	187	1,441	159	1,441	6,149	1,684	615	1,681	685	1,687
SQLite	401	11,331	191	11,331	137	11,331	3,618	11,565	313	11,584	687	11,568
MuJS	486	1,404	185	1,404	273	1,404	7,688	2,758	653	2,666	2,550	2,743
JSON-C	223	566	106	566	67	566	7,024	899	355	861	926	887

* The number *A(B)* in the table represents the number of path constraints (*No.pc*) collected in the first iterative run (A) and the line coverage (*Cover.*) achieved (B). We omitted the comparison with CSTG as it does not follow the iteration working style.

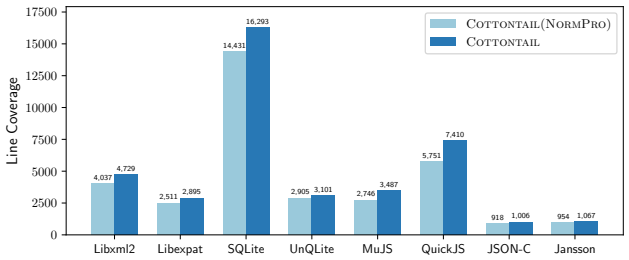

Figure 10. Results of normal and CoT prompts for constraint solving

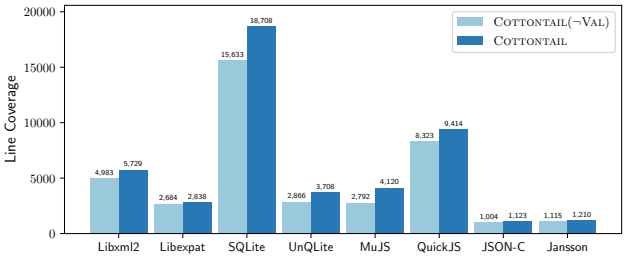

Figure 11. Comparison results of w/ or w/o test case validator

by-step reasoning enables the solver to navigate complex constraint spaces better. These results suggest that CoT prompts provide a significant advantage in guiding the model's symbolic reasoning process, leading to more effective exploration and ultimately higher coverage.

RQ2.3: How effective is the test case validator? Since it is critical to guarantee the soundness of test cases produced by concolic execution engines, we need to check if the validator designed in COTTONTAIL works. To have a fair comparison, we measure the line coverage achieved by COTTONTAIL and COTTONTAIL(¬VAL) (a variant approach of COTTONTAIL that removes the test case validator) under the same setting.

Figure 11 presents the line coverage achieved by COTTONTAIL and its variant COTTONTAIL(¬VAL) across various libraries. COTTONTAIL consistently outperforms or matches its counterpart, with significant improvements in larger libraries like SQLite, QuickJS, and Libxml2, where it achieves substantially higher line coverage. Overall, the results presented in the figure demonstrate that COTTONTAIL configuration is more effective, especially in complex codebases, highlighting the importance of test case validation and refinement for comprehensive coverage. This is reasonable as our refinement, designed in Algorithm 1 guarantees the newly generated test cases are expected to explore different program paths. Without the validator, COTTONTAIL(¬VAL) can be treated as a special variant of smart grey-box fuzzing without the guarantee of systematic program analysis.

We further evaluate the success rate of the LLM-driven constraint solver in directly producing correct solutions. Our results indicate that COTTONTAIL successfully solves 70.08% of the cases on average, with a failure rate of only 29.92%. Importantly, this low failure rate does not compromise the soundness of COTTONTAIL, as our newly designed validator—implemented via `refineTestCase` in Algorithm 1—systematically refines unreliable results.

RQ2.4: How effective is the history-guided seed acquisition? To have a better understanding of the contribution of seed acquisition, we carefully design the following variants:

- COTTONTAIL(RANDSEED): This variant performs random seed generation instead of history-guided generation.
- COTTONTAIL(INIT+¬SGEN): This variant is run with initial seed inputs and disables seed generation.
- COTTONTAIL(INIT+SGEN): This variant is run with initial seed inputs and generates new seeds when the test gets saturated (no increased coverage in three minutes).
- COTTONTAIL: This is the *default* version of our approach, which is run without any initial seed inputs, enabling the history-guided seed acquisition component.

We also include the baselines SYMCC and MARCO to provide a comprehensive comparison. We select one benchmark per format and run it for 12 hours to compare its line coverage. Figure 12 presents the detailed results.

Contribution of Guided Seed Generation. By comparing the results of COTTONTAIL with COTTONTAIL(RANDSEED), we can observe that the history-guided seed acquisition is superior to random seed generation. In all selected benchmarks, COTTONTAIL consistently outperforms its random-seed variant, achieving noticeably higher line coverage throughout the 12-hour window. For instance, in SQLite, COTTONTAIL reaches over 23,000 lines covered, whereas COTTONTAIL(RANDSEED) stalls below 19,000. These results demonstrate that historical execution feedback could guide seed acquisition significantly by prioritizing seeds with higher potential for new coverage.

Contribution for Changing Testing Saturation. We conduct

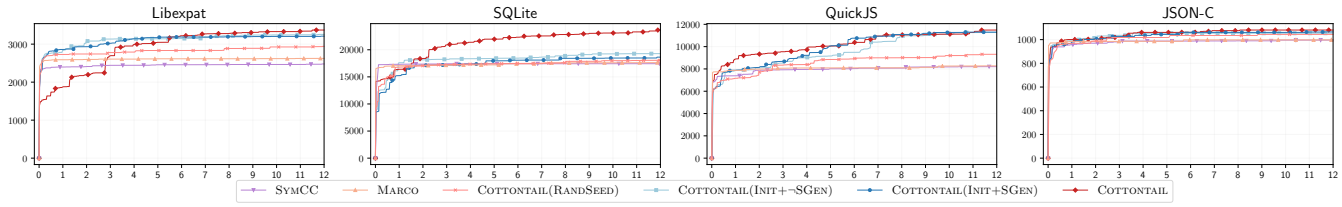

Figure 12. Line coverage comparison among CottonTail and variant approaches in 12 hours (*x-axis* indicates line coverage while *y-axis* the time)

two sets of comparative analyses to investigate it. First, by comparing CottonTail(Init+¬SGen) with CottonTail(Init+SGen), we can understand how this component boosts testing when the initial seeds are available. The results show that enabling seed generation significantly improves coverage when an initial seed is available, highlighting the importance of dynamic seed expansion. In particular, we can observe that baseline approaches usually get saturated within 2 hours, and it could be interesting to know how many lines can be covered after the saturation point. As a result, CottonTail(Init+SGen) covers 101, 376, 806, and 21 more lines over Libexpat, SQLite, QuickJS, and JSON-C than CottonTail(Init+¬SGen) within the latter 10 hours, indicating that enabling the seed generation will continuously increase the coverage, unlike saturating the seed generation. Second, by comparing CottonTail —including its variant configurations—with SymCC and Marco, we can find out how this component works when there are no seeds. The results show that CottonTail maintains superior performance even as coverage begins to saturate, demonstrating its effectiveness in exploring deeper program states under constrained conditions.

Answer to RQ2: By conducting carefully designed ablation studies, our results demonstrate the positive contribution of the newly designed components, including structure-aware constraint selection, LLM-driven constraint solving, and history-guided seed acquisition.

5.3. RQ3: Vulnerability Detection Capability

Details of Newly Detected Vulnerabilities. To evaluate the practical vulnerability detection capability of CottonTail, we run it (using the setting of the variant approach CottonTail(Init+¬SGen) for a fair comparison) and two baseline approaches in 12 hours and count the number of new vulnerabilities detected. During the experiments, CottonTail found 6 previously unknown vulnerabilities across three testing subjects and reported them to developers. The vulnerabilities with their subject, version, short description, and report status are listed in Table 4. These bugs involved heap memory leaks, buffer overflows, and stack overflows, with potential risks such as resource exhaustion, arbitrary code execution, or denial of services. Among the detected issues, 4 out of 6 have been fixed when submitting the paper (six new CVE IDs have been assigned), highlighting the practical impact of CottonTail in improving software security.

TABLE 4. New Vulnerabilities Detected

ID	Subject	Description	Status	CVE-Assigned
#1	MuJS	Memory leak	Fixed	CVE-2024-55061
#2	MuJS	Heap overflow	Fixed	CVE-2025-26082
#3	QuickJS	Stack overflow	Fixed	CVE-2024-13903
#4	QuickJS	Stack overflow	Fixed	CVE-2025-26081
#5	UnQLite	Global overflow	Reported	CVE-2025-26083
#6	UnQLite	Heap overflow	Reported	CVE-2025-3791

Comparison with Existing Approaches. Existing approaches failed or take too much time to detect it due to a structure-agnostic (or heavy) path constraints selection strategy or limited constraint-solving capabilities. To be specific, Marco can only detect vulnerability #5. Marco misses the other five vulnerabilities due to the limited path exploration and heavy scheduling on selecting nodes in CSTG. For example, when testing MuJS, we found that Marco takes 3.2 out of 12 (26.67%) hours of computing time to schedule and select an optimal path constraint for solving. SymCC can only detect four (#1, #3, #5, and #6) of them and misses others due to the aggressive path constraint elimination and restricted constraint-solving capabilities.

Case Study. To have a better grasp of the superiority of CottonTail, we present a case study. Figure 13(a) shows the vulnerable function, and Figure 13(b) shows seed and vulnerability triggering input generated by CottonTail. The issue occurs when `Ap_sort_cmp` (Line 2 in Figure 13(a)) analyzes the ill-defined comparator ("*function(x,y){return y;}*" shown in Line 4 in Figure 13(b)) in the vulnerability triggering input. The unexpected comparator causes `Ap_sort_cmp` to access invalid indices, i.e., *id_a* in the array during sorting. After invalid accessing, directly dereferencing the invalid pointer (*val_a*) leads to a heap overflow. In short, the unexpected return value from the `sort` function causes a heap overflow. Given the seed input[3], to find a new test input to trigger the overflow, a testing engine should construct an ill-defined `sort` function that returns an unexpected value. The efficient way is to negate the program constraint that requires changing the bytes after the 94^{th} byte '*r*' in `sort` function to a valid `return` statement that returns an unexpected value.

3. https://github.com/unifuzz/seeds/blob/master/general_evaluation/mujs/sort.js
4. https://github.com/ccxvii/mujs/issues/193.

```
// Application logic (buggy function) /* jsarray.c */
static int Ap_sort_cmp(js_State *J, int idx_a,int idx_b){
    js_Object *obj = js_tovalue(J, 0)->u.object;
    if (obj->u.a.simple) {
        js_Value *val_a = &obj->u.a.array[idx_a];
        js_Value *val_b = &obj->u.a.array[idx_b];
        int und_a = val_a->t.type == ...; // heap-overflow
    // ...
    }
}
```

(a) buggy function that triggers a new heap-overflow vulnerability[4] detected by COTTONTAIL.

```
// LLM generated test input
c = 30000; a = [];
for (i = 0; i < 2 * c; i += 1) {a.push(i%c);}
a.sort(function (x, y) { return y;}); print(a[100]);
```

(b) Seed input and vulnerability trigger generated by COTTONTAIL (the highlighted strings are from LLMs).

Figure 13. Vulnerable function and triggering input in case study

Due to structure-agnostic path constraints selection and limited constraint solving, MARCO [3] produced 26,575 test inputs (99.9% invalid) and failed to generate a trigger in 12 hours. SYMCC finds a trigger after 535^{th} iterations while SYMCC($\neg$MAP) after $1{,}811^{th}$ iterations of constraint solving. In summary, while existing concolic execution techniques can negate that branch, the resulting input is likely syntactically invalid and requires extra work by the concolic engine to pass the parser checks and generate the syntactically valid input. In contrast, benefiting from advanced structure-aware path constraint selection and smart constraint solving, COTTONTAIL detects this issue faster within only a few iterations (i.e., 55^{th}).

Answer to RQ3: COTTONTAIL is able to detect previously unknown vulnerabilities, showing a capable practical vulnerability detection capability.

6. Perspectives

Potential in Detecting Other Types of Bugs. We have shown that the highly structured test inputs could detect previously unknown memory-related vulnerabilities in RQ3. The test cases generated by COTTONTAIL could potentially detect other types of bugs (such as parsing or semantic bugs), benefiting from the higher passing rate for parsing checks. To detect more types of bugs, extra effort may be made to construct well-defined test oracles. To support our claim, we construct a simple test oracle by differential testing of JSON libraries to detect parsing issues. We define a potential bug as found if two parsers behave differently on the same test input. Since potential bugs can be false positives, as different parsers may be implemented in different standards (e.g., RFC 4627 for Jansson or RFC 7159 for `JSON-C`), new strategies must be applied to reduce such false positives caused by inconsistent standards. We manually analyzed a few of the potential issues and found a parsing bug[5] in JSON-C libraries. The bug is caused by an incomplete handling of control characters. Developers have confirmed and fixed the issue we reported.

5. https://github.com/json-c/json-c/issues/887

TABLE 5. TIME SPLIT FOR EXECUTION AND CONSTRAINT SOLVING

Time Split	SQLite		QuickJS	
	4o-mini	*4.1-nano*	*4o-mini*	*4.1-nano*
Execution Time (%)	9.55	12.12	10.11	11.90
Solving Time (%)	90.45	87.88	89.89	88.10

Potential in Practical Adoption. We believe COTTONTAIL can also have substantial potential to be applied in practical systematic white-box testing, such as SAGE [45], from the following four perspectives. *First*, the path constraints that are worth solving are significantly reduced. As shown in Table 3, the newly designed ECT-based path constraints selection can reduce many redundant path constraints (200%+ reduction), saving significant testing time in practice. *Second*, the cost of invocation of API is pretty low and COTTONTAIL can be easily integrated with both closed-source and open-source LLMs. We use the `gpt-4o-mini` as our base LLM, which is an affordable, cheap model ($0.150 / 1M tokens). Other LLMs such as `gpt-4.1-nano` (the most cost-effective gpt-4.1 model released on 14/04/2025) and `deepseek-v3` (a cheap and open source model released on 20/01/2025) can also be easily integrated within COTTONTAIL. *Third*, the potential of *Solve-Complete* paradigm for constraint solving is innovative and can be further improved via advanced solutions. During our experiments, we found that when using the CoT prompts, it would be more beneficial to combine expert knowledge into the completion phase. Our limited knowledge presented in Figure 2 has already shown a significant boost for high-demand structure-aware test input generation in the evaluation.

Fourth, we investigate the time spent on execution and constraint solving when running COTTONTAIL. The solving time refers to the time from the engine to take an input to perform concolic execution and give out the solution after constraint solving, and the execution time refers to the time for the execution of the test program with generated test inputs. Table 5 shows the results over two test programs across two LLMs within 1 hour of concolic testing. We could see that the solving time accounts for 89.08% of the overall testing time. This is mainly because the GPT API invocation takes time, and it is known that the LLMs are not as fast as traditional solvers like `Z3` [13], [27], [31]. Note that such a proportion is reasonable, as constraint solving is a complex process that requires significant computational resources and time [4], [32], [46]. For example, both SYMCC [1] and S2E [46] spend more than 90% of their solving time on analyzing complex software systems (e.g., OpenJPEG). Since practical performance is still bound to theoretical limits like constraint solving, further improvements (e.g., [47]) on speeding up API invocations could alleviate this issue, as we can already see that a newer version LLM `gpt-4.1-nano` could act faster compared with the older version of LLM.

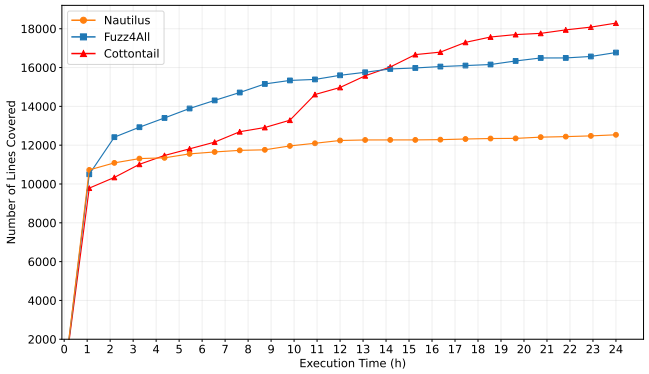

Figure 14. Comparison results of COTTONTAIL against Fuzz4All [29] and Nautilus [48] over `QuickJS` in 24 hours.

7. Discussion

Comparison with Structure-aware Black/Grey-box Fuzzing Approaches. In addition to our comparison with state-of-the-art white-box (concolic execution) approaches in the evaluation, we further showcase COTTONTAIL against complementary approaches like structure-aware black/grey-box fuzzing techniques. To do so, we evaluate COTTONTAIL against an LLM-based black-box fuzzer Fuzz4All [29] and a grey-box fuzzer Nautilus [48] over the same subject `QuickJS`. We select JavaScript as the target language since it has been employed in the evaluation of Nautilus and is a widely used, complex input format. Note that Fuzz4All does not support any format that Cottontail supports out of the box. Hence, we implemented additional support for JavaScript in Fuzz4All. we also used `gpt-4.1` (same model used in COTTONTAIL) and enabled an OpenAI key to set up the autoprompting in Fuzz4All. For a fair comparison, no initial seed inputs were given for any of the three approaches. We use line coverage (collected by `gcov`) to compare the performance of different approaches. We repeat fuzzing approaches five times and report median results.

The coverage results are shown in Figure 14. From the figures, we can observe that COTTONTAIL outperforms both Nautilus and Fuzz4All by achieving the highest code coverage when the experiment runs for 24 hours. We also notice that COTTONTAIL does not reach the highest coverage at the beginning, as it needs some time to set up concolic execution and gradually generate more test inputs. However, after a longer run, COTTONTAIL surpasses both baselines, after 4 hours for Nautilus and 14 hours for Fuzz4All. These results are expected, as although black-box and gray-box fuzzers could act more quickly to generate a large amount of test inputs, they often struggle to effectively explore more of the input space after a longer run (i.e., a saturation point tends to be reached) [41]. This is a long-standing challenge for black-box and grey-box fuzzing techniques [49], which tend to get stuck in local optima and fail to explore the input space effectively [40], [41]. In contrast, COTTONTAIL can not only systematically explore the input space but also leverage the power of LLMs to generate new seed inputs, covering more previously unexplored paths and eventually exploring more paths in the long run.

In summary, if the user has only a few hours of testing budget (e.g., 4 hours or less), black or grey-box fuzzing techniques may be more suitable. However, if the goal is to achieve high coverage over a longer period (more than 12 hours), COTTONTAIL would be a better choice.

API Costs for Running Experiments. The average invocation of GPT at 816 calls per subject, with an average cost of 0.78 USD per hour while using `gpt-4o-mini` model, demonstrating that the cost of using GPT APIs for constraint solving is relatively low. Traditional methods of constraint solving are limited by the solving capabilities as the aforementioned in previous sections; they produce a large amount of invalid test cases that have limited contribution to the testing effectiveness for generating highly structured test inputs, although they are faster. We believe the response time and robustness of LLM could be improved to further facilitate the test input generation capabilities.

Threats to Validity. Our findings and conclusions are subject to several potential threats to validity. The first concerns external validity, which relates to the generalizability of our results. As the subject of our study, we only selected SYMCC and MARCO and their variants, the state-of-the-art approaches for concolic execution. As objects of our study, we selected eight widely tested open-source libraries covering diverse domains, including XML, SQL, JavaScript, and JSON, which vary in size and popularity. While the subjects used in our evaluation are representative of a broad spectrum of real-world applications, we do not claim that the current COTTONTAIL applies to all software programs. For example, the current version does not include the evaluation over large software systems (e.g., V8 and GCC compilers). Such a scalability limitation is common in concolic execution, and we plan to integrate advanced techniques (e.g., selective path exploration [46]) to alleviate such a limitation. Another external validity concern driven by the use of LLMs is the risk of data leakage or memorization by LLMs. We believe this is unlikely, as the constraint-solving process in COTTONTAIL is unconventional and unique, making memorization improbable. The second threat involves internal validity, which refers to the extent to which the evidence supports the causal relationships claimed in our study. LLMs are known to exhibit the hallucination problem, generating outputs that may lack grounding in reality. However, COTTONTAIL addresses this issue by proposing a test case validator to validate and refine the generated test cases. To further reduce the influence of randomness, we also repeated each experiment five times.

Limitations. COTTONTAIL's effectiveness is limited by the completeness of the fuzzing driver. It is well-known that writing an effective fuzzing driver can be a challenging and time-consuming process. We plan to leverage the advanced technique [20] to mitigate the limitation in the future. As a source-code-based concolic execution, the current version of COTTONTAIL can only work for the test program whose source code is available. If only the binary of the target pro-

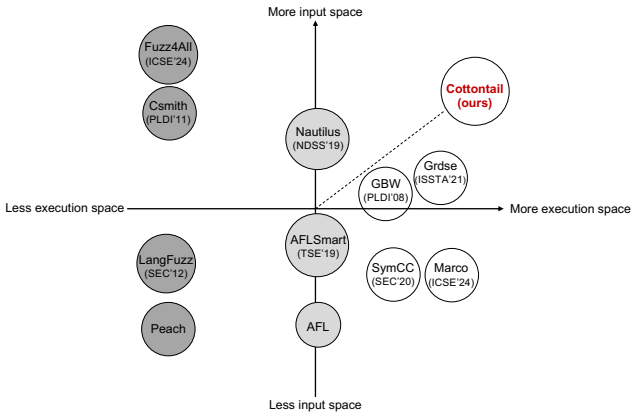

Figure 15. Positioning of COTTONTAIL in exploring input/execution space compared with existing works (the different colors represent three different fuzzing strategies: black-box, grey-box, and white-box fuzzing).

gram is available, our approach cannot be directly applied. We plan to further transfer the same idea to SYMQEMU [50], a binary concolic execution that shares the same idea of SYMCC, to alleviate the limitation. Another limitation is that it is unclear how COTTONTAIL will perform when handling programs that rely on unstructured inputs or formats unfamiliar to a pre-trained LLM, which we leave as future work to investigate.

8. Related Work

Traditional Fuzzing for Software Security. In past decades, many fuzzing techniques (including black, grey, and white-box based) have been proposed to improve software security. Essentially, they aim to explore the input space and execution space of a test program more effectively, where the *input space* refers to the set of all possible inputs that a program can take, and the *execution space* refers to the set of all possible execution paths that a program can run. To have a clear picture of the positioning of COTTONTAIL, we differentiate our work from existing fuzzing techniques in terms of the capability of exploring input space and execution space in Figure 15.

Black-box fuzzing approaches are typically limited in exploring both input space and execution space, as they do not have access to the internal structure or state of the program being tested. For example, Peach [51] applies format-aware mutations to an initial set of valid inputs using a user-defined input specification. Black-box grammar-based fuzzers focus on generating inputs that conform to specific syntactic structures, thereby improving the likelihood of covering more input space. For example, LangFuzz [52] and Grammarinator [53] parse existing regression tests using ANTLR grammars for structured input generation.

Greybox fuzzing improves upon black-box techniques by incorporating feedback mechanisms to guide the fuzzing process, which helps in exploring the execution space more effectively. AFL [54] is a well-known general-purpose fuzzing. Grey-box grammar-aware fuzzers increase the capabilities of exploring execution space. Superion [55] extends LangFuzz with coverage feedback, prioritizing mutated seeds that increase coverage. AFLSmart [56] addresses this issue by re-parsing each generated input added to the queue, ensuring structural integrity during fuzzing. Weizz [57] identifies fields and chunks within chunk-based file formats, and NestFuzz [58] infers inter-field dependencies and the hierarchical structure of inputs for better test case generation. While grey-box fuzzing techniques have made significant strides in exploring input spaces, they still face challenges in comprehensively exploring execution space.

Concolic execution is known for its capabilities to explore program paths systematically. QSYM [4] alleviates the strict soundness requirements of conventional concolic executors. Intriguer [59] further optimizes QSYM's symbolic execution with field-level knowledge. Angora [60] and Matryoshka [61] opt for taint analysis. SYMCC [1] first proposes compilation-based concolic execution to further gain performance enhancement. A recent work MARCO [3] explores code paths by decoupling branch flipping logic from the symbolic tracing logic and defers it until after all branch points uncovered are assessed. Syzspec [62] and Hulk [63] are two recent studies that also leverage input specifications to guide concolic execution for better path exploration. However, traditional concolic execution engines often struggle with generating valid inputs for programs that require highly structured inputs, as they typically do not incorporate knowledge about the input format or syntax. To address this challenge, grammar-based white-box fuzzing techniques have been proposed. Godefroid et al. [64] introduced a grammar-based white-box fuzzing approach, which advocates for the use of token symbolization during symbolic execution. The resulting token constraints are then solved using the input grammar. Similarly, CESE [65] utilizes an input grammar to improve dynamic symbolic execution for programs that parse this grammar. Grdse [66] propose grammar-agnostic dynamic symbolic execution that automatically infers input grammars from seed inputs. Although promising, they are still limited in exploring complex input formats and deep program paths.

Compared with existing fuzzing approaches, we position COTTONTAIL as a novel LLM-driven concolic execution engine that is able to effectively explore both input space and execution space, as shown in Figure 15. Compared with black/grey-box fuzzing, COTTONTAIL performs systematic path exploration instead of random test case generation/mutation, making it more suitable for comprehensive program analysis. Compared to white-box fuzzing, COTTONTAIL is superior for its *structure-aware path selection*, *smart constraint solving*, and *capable of acquiring new seeds*, which address three long-standing issues in existing concolic execution approaches. The major contribution, i.e., smart constraint solving that leverages LLM with *Solve-Complete* paradigm, is orthogonal to existing white-box fuzzing techniques, which can be potentially integrated to further improve the performance of concolic execution.

LLM-assisted Fuzzing for Software Security. Recent research has demonstrated their potential in software secu-

rity tasks such as fuzz testing. ChatFuzz [31] employs an LLM to enhance input generation for protocol fuzzing. Codamosa [67] uses LLMs to generate example test cases for under-covered functions, addressing coverage plateaus in search-based software testing. Fuzz4All [29] combines LLMs with evolutionary algorithms to generate structured test inputs for programs C/C++. CovRL-Fuzz [68] and InputBlaster [69] integrate LLMs to enhance input generation for fuzzing in JavaScript engines and mobile apps. AutoExe [70] uses a generic code-based representation and performs program synthesis to generate test cases.

Beyond test input generation, LLMs are also applied to generate fuzz drivers for APIs. Oss-fuzz-gen [71], [72] employs few-shot learning techniques to create fuzz drivers based on given APIs. Promptfuzz [20] generates fuzz drivers through various API mutations, and Zhang et al. [73] evaluate different strategies for LLM-based driver generation. *TitanFuzz* [30] leverages LLMs to generate both harness programs and arguments for fuzzing deep learning libraries.

Unlike existing LLM-based solutions that randomly generate test inputs, which can explore a larger input space but are limited in exploring execution space (e.g., Fuzz4All [29] as shown in Figure 15), we propose to combine more precise semantic information (i.e., path constraints) with LLM to improve its test case generation capabilities. Besides, we also utilize a novel *Solve-Complete* paradigm for smart constraint solving, yielding promising code coverage and vulnerability detection capabilities.

9. Conclusion

We presented COTTONTAIL, a new LLM-driven concolic execution engine to generate highly structured test inputs for parsing testing. COTTONTAIL's novelties lie in the design of structure-aware constraint selection to select path constraints that are worth exploring, LLM-driven constraint solving to smartly produce test cases that not only satisfy the path constraints but also align with syntax rules, and history-guided seed acquisition to generate new seed inputs whenever the engine starts testing or the testing process saturates. We compared COTTONTAIL with state-of-the-art concolic execution engines, and the results demonstrate the superior performance of COTTONTAIL in terms of code coverage and vulnerability detection capability. Our study has shown promising potential in combining traditional program analysis with LLMs, calling for more advanced proposals combining LLMs to improve software security.

Acknowledgments

We sincerely appreciate Cristian Cadar for his constructive suggestions in improving the article. We thank the anonymous reviewers and our shepherd for their insightful feedback and comments. We also appreciate the developers of `MuJS`, `QuickJS`, `UnQlite`, and `JSON-C` for their prompt confirmation and fixing of our reported issues.

References

[1] S. Poeplau and A. Francillon, "Symbolic Execution with SymCC: Don't Interpret, Compile!" in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 181–198.

[2] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, "SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, 2022, pp. 2531–2548.

[3] J. Hu, Y. Duan, and H. Yin, "Marco: A Stochastic Asynchronous Concolic Explorer," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024, pp. 1–12.

[4] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security)*, 2018, pp. 745–761.

[5] H. Tu, "Boosting Symbolic Execution for Heap-Based Vulnerability Detection and Exploit Generation," in *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings (ICSE-NIER)*, 2023, pp. 218–220.

[6] P. Pitigalaarachchi, X. Ding, H. Qiu, H. Tu, J. Hong, and L. Jiang, "KRover: A Symbolic Execution Engine for Dynamic Kernel Analysis," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023, pp. 2009–2023.

[7] H. Tu, L. Jiang, X. Ding, and H. Jiang, "FastKLEE: Faster Symbolic Execution via Reducing Redundant Bound Checking of Type-Safe Pointers," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 1741–1745.

[8] S. Lu, Z. Feng, D. Guo, S. Wang, D. Tang, N. Duan, M. Zhou *et al.*, "CodeXGLUE: A Benchmark Dataset and Open Challenge for Code Intelligence," *arXiv preprint arXiv:2102.04664*, 2021.

[9] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an LLM to Help with Code Understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[10] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen, "Source Code Summarization in the Era of Large Language Models," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 1882–1894.

[11] W. Ma, S. Liu, Z. Lin, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, L. Li, and Y. Liu, "LMs: Understanding Code Syntax and Semantics for Code Analysis," *arXiv preprint arXiv:2305.12138*, 2023.

[12] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, Asmita, R. Tsang, N. Nazari, H. Wang, and H. Homayoun, "Large Language Models for Code Analysis: Do LLMs Really Do Their Job?" in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, 2024, pp. 829–846.

[13] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought Prompting Elicits Reasoning in Large Language Models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS)*, 2022, pp. 1–14.

[14] D. of CGC. DARPA Cyber Grand Challenge. [Online]. Available: https://www.darpa.mil/research/programs/cyber-grand-challenge

[15] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012, pp. 380–394.

[16] D. of the Team Atlanta. Concolic Execution used in the Winner Team of AIxCC. [Online]. Available: https://github.com/Team-Atlanta/aixcc-afc-atlantis/tree/main/example-crs-webservice/crs-multilang/uniafl/src/concolic

[17] Z3. (2025) A Theorem Prover from Microsoft Research. [Online]. Available: https://github.com/z3prover/z3

[18] D. Trabish, S. Itzhaky, and N. Rinetzky, "A bounded symbolic-size model for symbolic execution," in *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1190–1201.

[19] H. Tu, L. Jiang, J. Hong, X. Ding, and H. Jiang, "Concretely Mapped Symbolic Memory Locations for Memory Error Detection," *IEEE Transactions on Software Engineering*, vol. 50, no. 7, pp. 1747–1767, 2024.

[20] Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Prompt Fuzzing for Fuzz Driver Generation," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3793–3807.

[21] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 2271–2287.

[22] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, "Utopia: Automatic generation of fuzz driver using unit tests," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2023, pp. 2676–2692.

[23] J. Lin and D. Mohaisen, "From Large to Mammoth: A Comparative Evaluation of Large Language Models in Vulnerability Detection," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025, pp. 1–18.

[24] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2023, pp. 2339–2356.

[25] Z. Luo, H. Zhao, D. Wolff, C. Cadar, and A. Roychoudhury, "Agentic Concolic Execution," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2026, pp. 1–19.

[26] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2024, pp. 862–880.

[27] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie, "A Survey on Evaluation of Large Language Models," *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 3, 2024.

[28] Y. Nong, M. Aldeen, L. Cheng, H. Hu, F. Chen, and H. Cai, "Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities," 2024. [Online]. Available: https://arxiv.org/abs/2402.17230

[29] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4All: Universal Fuzzing with Large Language Models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[30] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models," in *Proceedings of the ACM International Symposium on Software Testing and Analysis*, 2023, pp. 423–435.

[31] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large Language Model Guided Protocol Fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024, pp. 1–15.

[32] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 209–224.

[33] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings of the Network and Distributed System Security Symposium (NSDI)*, 2016, pp. 1–16.

[34] S. Shankar, J. Zamfirescu-Pereira, B. Hartmann, A. Parameswaran, and I. Arawjo, "Who Validates the Validators? Aligning LLM-assisted Evaluation of LLM Outputs with Human Preferences," in *Proceedings of the Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2024, pp. 1–14.

[35] Z. Wang, L. Zhang, C. Cao, N. Luo, X. Luo, and P. Liu, "How Does Naming Affect LLMs on Code Analysis Tasks?" 2024. [Online]. Available: https://arxiv.org/abs/2307.12488

[36] R. S. Alsuhaibani, C. D. Newman, M. J. Decker, M. L. Collard, and J. I. Maletic, "On the naming of methods: A survey of professional developers," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 587—599.

[37] D. G. Feitelson, A. Mizrahi, N. Noy, A. B. Shabat, O. Eliyahu, and R. Sheffer, "How Developers Choose Names," *IEEE Transactions on Software Engineering*, vol. 48, no. 01, pp. 37–52, 2022.

[38] C. Charitsis, C. Piech, and J. C. Mitchell, "Function names: Quantifying the relationship between identifiers and their functionality to improve them," in *Proceedings of the Ninth ACM Conference on Learning@ Scale*, 2022, pp. 93–101.

[39] W. Akram, Y. Jiang, Y. Zhang, H. A. Khan, and H. Liu, "LLM-Based Method Name Suggestion with Automatically Generated Context-Rich Prompts," *Proceedings of the ACM Software Engineering*, vol. 2, no. FSE, pp. 1–22, 2025.

[40] W. Gao, V.-T. Pham, D. Liu, O. Chang, T. Murray, and B. I. Rubinstein, "Beyond the Coverage Plateau: A Comprehensive Study of Fuzz Blockers (Registered Report)," in *Proceedings of the 2nd International Fuzzing Workshop*, 2023, pp. 47–55.

[41] T. Klooster, F. Turkmen, G. Broenink, R. Ten Hove, and M. Böhme, "Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines," in *IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, 2023, pp. 25–32.

[42] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proceedings of the USENIX Annual Technical Conference*, 2012, pp. 309–318.

[43] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path Sensitive Fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 679–696.

[44] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[45] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[46] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[47] K. Ayoub. Accelerating Large Language Models with TensorRT-LLM and Serving (OpenAI-Compatible API). https://blog.gopenai.com/accelerating-large-language-models-with-tensorrt-llm-db928323ddbf.

[48] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019, pp. 1–15.

[49] M. Boehme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and Reflections," *IEEE Software*, vol. 38, no. 03, pp. 79–86, 2021.

[50] S. Poeplau and A. Francillon, "SymQEMU: Compilation-based Symbolic Execution for Binaries," in *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2021, pp. 1–18.

[51] Peach Fuzzer. [Online]. Available: https://peachtech.gitlab.io/peach-fuzzer-community/

[52] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *Proceedings of the 21st USENIX Security Symposium (USENIX Security)*, 2012, pp. 445–458.

[53] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: A Grammar-based Open Source Fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 45–48.

[54] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020, pp. 1–12.

[55] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware Greybox Fuzzing," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019, pp. 724–735.

[56] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart Greybox Fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2019.

[57] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic Greybox Fuzzing for Structured Binary Formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020, pp. 1–13.

[58] P. Deng, Z. Yang, L. Zhang, G. Yang, W. Hong, Y. Zhang, and M. Yang, "NestFuzz: Enhancing Fuzzing with Comprehensive Understanding of Input Processing Logic," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023, pp. 1272–1286.

[59] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level Constraint Solving for Hybrid Fuzzing," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2019, pp. 515–530.

[60] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 711–725.

[61] P. Chen, J. Liu, and H. Chen, "Matryoshka: Fuzzing Deeply Nested Branches," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 499–513.

[62] Y. Hao, J. Pu, X. Li, Z. Qian, and A. A. Sani, "SyzSpec: Specification Generation for Linux Kernel Fuzzing via Under-Constrained Symbolic Execution," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2025, pp. 1–14.

[63] Z. Liu, T. Lee, J. Yu, Z. Kang, and Y. Cao, "The DOMino Effect: Detecting and Exploiting DOM Clobbering Gadgets via Concolic Execution with Symbolic DOM," in *Proceedings of the 34th USENIX Security Symposium (USENIX Security)*, 2025, pp. 8293–8312.

[64] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based Whitebox Fuzzing," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2008, pp. 206–215.

[65] R. Majumdar and R.-G. Xu, "Directed Test Generation Using Symbolic Grammars," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ICSE)*, 2007, pp. 134–143.

[66] W. Pan, Z. Chen, G. Zhang, Y. Luo, Y. Zhang, and J. Wang, "Grammar-Agnostic Symbolic Execution by Token Symbolization," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 374–387.

[67] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023, pp. 919–931.

[68] J. Eom, S. Jeong, and T. Kwon, "Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation," in *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1656–1668.

[69] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Z. Tian, Y. Huang, J. Hu, and Q. Wang, "Testing the Limits: Unusual Text Inputs Generation for Mobile App Crash Detection with Large Language Model," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–12.

[70] Y. Li, R. Meng, and G. J. Duck, "Large Language Model powered Symbolic Execution," vol. 9, no. OOPSLA2, 2025, pp. 1–29.

[71] J. M. Dongge Liu and O. Chang. Fuzz Target Generation Using LLMs. https://google.github.io/oss-fuzz/research/llms/target_generation/.

[72] Google, "oss-fuzz-gen," https://github.com/google/oss-fuzz-gen.

[73] C. Zhang, Y. Zheng, M. Bai, Y. Li, W. Ma, X. Xie, Y. Li, L. Sun, and Y. Liu, "How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1223–1235.

Appendix A. Additional Experiments

A.1. Results of Running Setting 2 in Section §5.1

Our results show that COTTONTAIL consistently achieves the highest line and branch coverage in the majority of the evaluated benchmarks, although it was allocated only one hour of execution time - significantly less than the 12-hour time budget used by baseline approaches such as SYMCC and MARCO (and their respective variants). In particular, COTTONTAIL attains the best line coverage in 7 of 8 programs. On average, it covers up to 4.4k lines (6.77% on average) more and 2.8k branches (9.57% on average) more in total than all other techniques. These results highlight the effectiveness and efficiency of COTTONTAIL in rapidly exploring diverse program paths and uncovering deep execution behaviors, even under constrained time settings. There are a few cases where COTTONTAIL covers less branch coverage than MARCO (e.g., in `Libxml2`). This is because MARCO designs the path selection based on random sampling, so a few more line/branch coverages are expected.

A.2. Results of Running Setting 3 in Section §5.1

When comparing COTTONTAIL against the baseline approaches SYMCC, MARCO, and their variants within 12 hours, the results show that COTTONTAIL performs better than *all* comparative approaches and achieves higher line coverage rates from 15.10% to 21.41% on average. Note that we changed the base model from `gpt-4o` to `gpt-4o-mini` in this setting, as `gpt-4o` is already superior to baseline approaches in one hour, and `gpt-4o-mini` is more cost-effective.

It is worth noting that on many benchmarks (e.g., `Libexpat` and `SQLite`), both SYMCC and MARCO exhibit early saturation in their coverage progress. For example, SYMCC quickly reaches a plateau in one hour and then shows minimal improvement, indicating limited ability to uncover additional program behaviors beyond its initial exploration. The final coverage achieved by both techniques remains substantially lower than that of other approaches, suggesting that their underlying strategies are less effective in sustaining exploration over time. Thus, fresh seeds are required to change the saturation and make the testing more effective, which motivates us to design a new seed generation strategy to help explore more paths.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper presents COTTONTAIL, a novel concolic execution framework to overcome the several key limitations (i.e., structure-agnostic constraint selection, syntax-ignorant solving, and reliance on manual or random seed inputs) in the existing concolic execution systems, by leveraging Large Language Models (LLMs).

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long Known Issue
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The use of LLMs as integral components in concolic execution is innovative.
2) The paper tackles several long-standing issues in concolic execution, and the experiments show that the proposed solution indeed addresses these issues effectively.
3) The prototype COTTONTAIL is open-sourced to enable future science. It not only allows further improvement on concolic execution, but also other software security research that uses concolic execution as an analysis tool.

B.4. Noteworthy Concerns

1) The paper's evaluation targets are limited to small programs that rely on structured input formats, which are well understood by LLMs. Its performance on large programs, programs with unstructured inputs, and programs with input formats unknown by LLMs is not evaluated.
2) While the Discussion and Related Work carefully place the proposed technique into a broader context (i.e., automated testing techniques, including fuzzing), such contextualization comes late in the paper – those who only read the introduction and evaluation may fail to appreciate the technique's limitations.