Concrete Constraint Guided Symbolic Execution

Yue Sun[1], Guowei Yang[2], Shichao Lv[1], Zhi Li[1], Limin Sun[1*]
[1]Institute of Information Engineering, CAS; School of Cyber Security, UCAS, China
[2]The University of Queensland, Australia
{sunyue0205,lvshichao,lizhi,sunlimin}@iie.ac.cn,guowei.yang@uq.edu.au

ABSTRACT

Symbolic execution is a popular program analysis technique. It systematically explores all feasible paths of a program but its scalability is largely limited by the path explosion problem, which causes the number of paths proliferates at runtime. A key idea in existing methods to mitigate this problem is to guide the selection of states for path exploration, which primarily relies on the features to represent program states. In this paper, we propose concrete constraint guided symbolic execution, which aims to cover more concrete branches and ultimately improve the overall code coverage during symbolic execution. Our key insight is based on the fact that symbolic execution strives to cover all symbolic branches while concrete branches are neglected, and directing symbolic execution toward uncovered concrete branches has a great potential to improve the overall code coverage. The experimental results demonstrate that our approach can improve the ability of KLEE to both increase code coverage and find more security violations on 10 open-source C programs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Symbolic Execution, Data Dependency Analysis

ACM Reference Format:
Yue Sun[1], Guowei Yang[2], Shichao Lv[1], Zhi Li[1], Limin Sun[1*]. 2024. Concrete Constraint Guided Symbolic Execution. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, Lisbon, Portugal, 12 pages. https://doi.org/10.1145/3597503.3639078

1 INTRODUCTION

Symbolic execution is a prevalent and powerful technique originated in the 70's for program testing and debugging [14, 25]. Over the years, researchers worldwide have made significant efforts to enhance the performance and scalability of open-source symbolic execution tools. This technique has found applications in various academic fields, including software testing [26, 33, 34, 54], program verification [17, 23, 40], vulnerability analysis [47, 48], and firmware emulation [9, 24, 57]. In addition, industrial practices are also tried in combination with symbolic execution for enhancing software security in various domains such as Baidu [28], Cyberhaven [13], Fujitsu [19] and Trail of Bits [21].

The concept behind classical symbolic execution is to replace concrete inputs with symbolic inputs in order to explore all feasible paths within a program [7]. During symbolic execution, program variables are mapped to symbols, and the constraints formed by these symbols are collected along each path. These constraints are then utilized by SMT solvers to find concrete values for for program variables to explore new paths [8]. However, this approach generates new states whenever symbolic branching conditions are encountered, leading to a surge in the number of states and resulting in the problem known as *path explosion* [4]. Consequently, significant time and memory resources are wasted without an efficient mechanism to choose promising states. To address this issue, it is crucial to represent each state using static and dynamic features specific to the program. These features can assist in selecting promising states [53], pruning [12] or merging redundant states [27]. Therefore, careful selection of features is essential to enhance the efficacy of path exploration.

Existing methods. The features to represent program states serve as a proxy to find new program paths by guiding states to favor specific program properties. For instance, existing methods select features such as path depth, query time, sub-path, the number of executed instructions and generated test cases and so on [7, 29]. However, the search strategy based on one single feature is not adaptive to all programs since they are largely distinct in the design and implementation of programs such as code structure, branching conditions and the dependencies between program variables [11, 22]. Therefore, recent works have shifted to using machine learning techniques to model several manually picked features to learn a more comprehensive and robust search strategy for better state selection [10, 11, 22, 37]. Yet, it is difficult for machine-learning based methods to achieve good performance for all programs on software testing due to the lack of guidance on choosing suitable models [45]. In addition, machine-learning based methods are also limited to the training data and hyper-parameters.

New insight. In general, the variables in branching conditions during symbolic execution can either be symbolic or concrete [5]. When all the variables in a branching condition are concrete, this branching condition is considered as a *concrete branching condition*, e.g., *x>1* when the variable *x* is concrete. Furthermore, we denote the branching constraint for each branch of this branching condition as *concrete constraint*. In this paper, we propose to represent each program state based on its ability to cover more concrete branches. The idea derives from the fact that enormous efforts have been made on constraint solving techniques to deal with *symbolic branching*

*Corresponding Author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639078

conditions [3, 38, 46, 51]. However, according to our observations in Section 2, *concrete branching conditions* take up over 95% during symbolic execution but nearly 70% of them are partially covered, i.e., only one branch of the branching condition is covered, in real-world programs. Therefore, there is a great potential to enhance symbolic execution to fully cover more *concrete branching conditions* to improve overall code coverage.

Furthermore, for a partially covered concrete branching condition, it is necessary for the states reaching it to carry different values for the variables involved in the branching condition, so that the constraint for the uncovered branch can be satisfied. However, existing symbolic execution tools face a challenge in this regard. They often rely on other features for state selection while overlooking the correlation between the definitions of these variables and the selection of promising states. Since these variables are usually defined and used at many different program locations, the execution path that only covered one or several definitions have a rather low possibility to satisfy the branching constraints of the uncovered branches. Consequently, these tools frequently waste time in feeding *incorrect* definitions of these variables in uncovered concrete branching conditions. This not only reduces the chance of covering more concrete branches but also reduces the chance of further covering more symbolic branches, and thus decreases the overall efficacy of path exploration. In this paper, we shift the focus to the variables in concrete branching conditions to investigate the states that have defined them, and whether these definitions can satisfy specific concrete constraints to select more promising states to improve overall code coverage.

Our solution. Overall, we leverage the inter-procedural data dependency of the variables in concrete branching conditions as the core feature for state selection during symbolic execution. By identifying *when a state defines a variable in a partially covered concrete branching condition*, we can prioritize this state *if this definition can satisfy the concrete constraint of the uncovered branch of the branching condition.* However, there are two main challenges: one, extracting the inter-procedural data dependency of program variables can be expensive; two, it is not clear how to leverage data dependency for the state selection during symbolic execution. Existing inter-procedural static value-flow analysis methods for program variables are mostly expensive in terms of balancing accuracy and performance [39]. Instead, we propose a backward data dependency analysis method to identify the source variables of a given target variable. Then, we use these source variables as intermediate variables to establish the inter-procedural data dependency. During symbolic execution, we prioritize states that have specific variable definitions that can satisfy the constraint corresponding to the uncovered branch of the partially covered branching condition.

Evaluation. We implement our method on top of a widely-used open-source symbolic execution framework, KLEE [7, 36]. To evaluate the efficiency, we use 10 open-source C programs as benchmarks, which are also used in relevant works on symbolic execution. For code coverage, our method improves the branch coverage by 28.8% on average compared to the best of existing methods and fully covers 15.6 more concrete branching conditions than existing methods. In addition, our method is stable with a low sensitivity on the settings of hyper-parameters for optimizations. Furthermore, our method triggers 6 out of 7 security violations in 5 programs while other methods trigger at most 3 using the generated test cases.

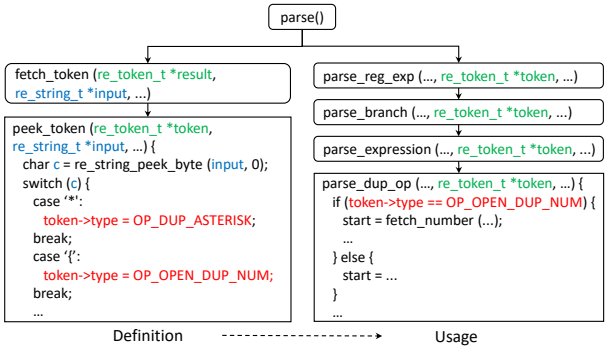

Figure 1: Function call chains for parsing inputs in *grep 3.6*.

Contributions. We summarize our main contributions as follows:

- We design an algorithm to extract the inter-procedural data dependencies for the variables involved in branching conditions.
- We propose a concrete constraint guided search strategy to cover more concrete branches to improve the overall branch coverage for symbolic execution.
- We implement and evaluate our method on 10 open-source C programs on branch coverage and security violations, which demonstrates the effectiveness of our method. We make our tool publicly available at https://github.com/XMUsuny/cgs.

2 MOTIVATION

In this section, we use the source code from the subject program *grep 3.6* [1] to present an example of concrete constraint, and then analyze the branching conditions encountered in symbolic execution on real-world programs to motivate this work.

An example of concrete constraint. We illustrate the call chains from *grep 3.6* to demonstrate its use for text matching with regular expressions in Figure 1. After receiving user inputs from command line, the `parse` function calls the two functions on the left to assign a value to *token->type* and then proceeds to call the four functions on the right to use this definition in the branching condition of the *if* statement. Specifically, the `peek_token` function assigns a concrete value to *token->type* based on the *input* variable and the `parse_dup_op` function employs different handlers to interpret tokens depending on *token->type*. On this call chain, the struct-typed pointer *token* stays unchanged as the function parameter and is directly passed. This establishes an inter-procedural data dependency between the definitions and usages of *token->type* in these functions. It is important to note that the branching constraint in the `parse_dup_op` function is always a concrete constraint because *token->type* is consistently assigned a concrete value during symbolic execution. Consequently, a single state can only satisfy one concrete constraint and partially cover the corresponding concrete branch of this branching condition.

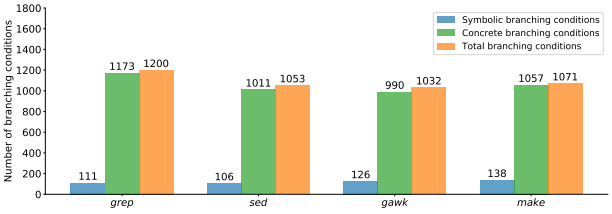

Figure 2: Number of branching conditions in different types.

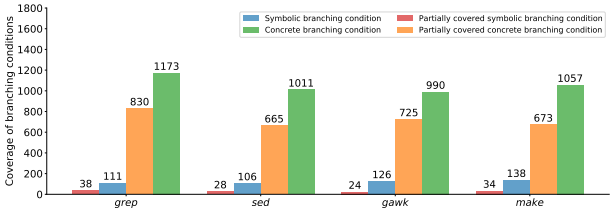

Figure 3: Coverage of different branching conditions.

Branch coverage at runtime. To investigate how the symbolic/concrete branches are covered during symbolic execution, we conducted an experiment where we ran four typical real-world programs for 2 hours using KLEE, employing a random-path search strategy [7]. We gathered the number of branching conditions in different types, and their coverage in Figure 2 and Figure 3, respectively. For simplicity, we only focused on the conditional statements with LLVM *icmp* instructions as branching conditions. Note that there can be roughly two categories of concrete branching conditions: (1) they are originated from different input sources that are not symbolic (e.g., environment variables and configuration files). (2) they are constrained by the variables that are implicitly affected by symbolic inputs as shown in Figure 1. Intuitively, the second one takes the major role so our experiments does not consider the influence of the first one.

According to the results in Figure 2, the concrete branching conditions accounted for over 95% of the total branching conditions. Furthermore, we find some branching conditions can be both symbolic and concrete since the variables involved in these branching constraints may be defined using concrete values along some paths while using symbolic values along other paths. Consequently, their sum exceeds the total number of branching conditions.

Moreover, we collected the coverage of branching conditions in different types, and summarised the results in Figure 3. Our analysis revealed that, nearly 25% of the symbolic branching conditions were partially covered, i.e., only one branch of the branching condition is covered, while nearly 70% of the concrete branching conditions were partially covered. In summary, the results suggest a significant disparity in the coverage of branching conditions with different branching constraints. Concrete branching conditions showed a higher rate of being partially covered compared to symbolic ones.

Concrete constraint guided search strategy. Since concrete branching conditions encompass a significant majority and a large portion of them are only partially covered. we believe there is a great potential to improve overall code coverage by guiding symbolic execution towards covering more concrete branches that are reached but not covered during symbolic execution.

In this paper, we leverage static analysis to soundily extract the inter-procedural data dependency of the variables in branching conditions. Moving forward, by analyzing the uncovered concrete branching conditions during symbolic execution, we can identify the states that have defined these variables with values that can satisfy the unsatisfied concrete constraints. Finally, we can leverage this information to steer symbolic execution to cover more concrete branches.

3 DESIGN

In this paper, we focus on covering more concrete branches during symbolic execution using the data dependency of the variables in concrete branching conditions. Respectively, the branch dependency analysis module aims to statically extract inter-procedural data dependency of these variables. The concrete constraint guided searcher uses the extracted data dependency for efficient state selection during symbolic execution.

3.1 Branch Dependency Analysis

This module aims to construct the inter-procedural data dependency between the branching conditions and the definitions of the variables used by them. Respectively, we first identify the source variables for each variable used in the branching conditions and the address of assignment instructions. Then, we match these source variables based on the values or types of them to construct the inter-procedural data dependency,

Source variable definition. Given a program variable, we define its source variable as the variable from which the value is initially assigned within a function. We describe each source variable with type and value. In general, the type of source variable within a function falls in the following types: *global variable*, *local variable* and *function parameter*. The value of source variable depends on its type. For global and local variable, the value is explicitly defined with unique instruction identifier. However, for function parameter in inter-procedural scope, the values may come from different instances so they are ambiguous. In addition, since some function parameters always remain unchanged across functions, we further divide the type of function parameter into three subtypes: *non-pointer parameter*, *struct-typed pointer parameter* and *non-struct-typed pointer parameter*.

The *non-pointer parameter* is a non-pointer variable that is usually defined in the caller functions on function call graph. The *struct-typed pointer parameter* is a struct-typed pointer variable that is usually defined and used in different functions but always stays unchanged. As shown in Figure 1, it can be an intermediate variable to build an inter-procedural data dependency between the variables in different functions. Moreover, sometimes there is more than one nested struct-typed variables to reach sink variable, which builds a list of struct-typed pointers and each node is denoted by the type and offset in nested structs. For instance, the source variable of *dfa->nodes_alloc* is recorded as *[(regex_t *, 0), (re_dfa_t *, 1)]* in Listing 1. In addition, we do not consider *non-struct-typed pointer parameter* such as string and integer pointers.

```
1
2  static reg_errcode_t
3  analyze(regex_t *preg) {
4    re_dfa_t *dfa = preg->buffer;
5    dfa->nexts = re_malloc (Idx, dfa->nodes_alloc);
6    ..
7  }
```

Listing 1: An example of struct-typed pointer variable

Source variable identification. We propose an algorithm for performing a backward data dependency analysis. The goal of this algorithm is to identify the source variables of a given variable and categorize them into one of the five types mentioned earlier, which we refer to as variable v in Algorithm 1. In general, our approach involves tracking LLVM-IR instructions stored in Q_{inst} in a depth-first manner, going backward along the def-use chains for different types of instructions. Here are the key steps we follow: ❶ When encountering invalid instructions in lines 7-8 (e.g., *call, switch,* and *icmp*), we stop tracking since we only focus on data dependency. ❷ For *gep* instruction, we record the type and offset of the accessed structure member in a struct-typed data in P_{st}, and add the instruction with the type of struct-typed pointer to Q_{inst} in line 9-11. ❸ For other instructions, we employ different strategies for tracking based on the type of each operand in current instruction.

Specifically, ① If the operand is a global variable, we directly add it as the source variable in line 14-15. ② If the operand is a local variable, we first use the `getStores` function to find all the *store* instructions that defines this local variable in line 17. Since local variable can be defined using function parameter at the function entry, we use the `findParameter` function to find a definition that comes from a function parameter in line 18. If this function parameter is non-pointer or struct-typed pointer, we create new source variable in line 19-20 and use the `addSource` function to set the type of this source variable to P_{st} and clear it. ③ If no function parameter is found, we add this local variable as the source variable in line 22. Moreover, we use the `selectInsts` function to find new store instruction for the next round in line 23. We first attempt to find the store instruction that dominates current instruction to Q_{inst}. If not found, we adds the nearest store instruction defines this local variable to Q_{inst} in line 25 if it can reach v on the control flow graph of current function and the written data is an instruction. ④ Otherwise, if the operand is neither a global or local variable, we add it to Q_{inst} if it is an instruction in line 27-28 for the next round.

Example. We illustrate the workflow to identify the source variable of variable %41 in the upper part in Figure 4. The instructions in blue color are the traced instructions in Q_{inst}. In line 63, we add struct-typed element *[(re_token_t *, 1)]* to P_{st}. For local variable %11, we find the *store* instruction in line 26 and the written data %3 is a function parameter. Eventually, we set %3 as the source variable of variable %41 in type of *struct-typed pointer parameter*.

Source variable for usage. The usage is the value of branching condition. For branch statements, we consider all the *switch*, and partial *br* instructions which use the results of *icmp* instructions as the branching conditions. We extract these variables from branch statements and then use Algorithm 1 to extract the source variables of them. For instance, we identify the source variable of variable %41 in the *br* instruction in line 68 as parameter %3 in Figure 4 for the `parse_dup_op` function in Figure 1.

Algorithm 1: Source variable identification

Input: An instruction that contains a sink variable (v)
Output: A set of source variables (S_v)

```
1  Function ExtractSourceVars(v):
2      S_v ← ∅ ;                          ▷ set of source variables
3      P_st ← { } ;                       ▷ list of struct-typed data
4      Q_inst ← {v} ;                     ▷ list of instructions to track
5      while Q_inst ≠ ∅ do
6          inst ← Q_inst.pop();
7          if inst ∈ {call, switch, icmp} then
8              continue;
9          else if inst is a gep instruction that accesses struct then
10             P_st.push((gep.type, gep.offset));
11             Q_inst.push(gep.pointer);
12         else
13             for op ∈ inst.operands do
14                 if op is a global variable g then
15                     S_v ← S_v ∪ addSource(g, ∅);
16                 else if op is a local variable l then
17                     store_l ← getStores(l);
18                     param ← findParameter(store_l);
19                     if param is non- or struct-typed- pointer then
20                         S_v ← S_v ∪ addSource(param, P_st);
21                     if no parameter is found then
22                         S_v ← S_v ∪ addSource(l, ∅);
23                         store ← selectInst(store_l);
24                         if store.value is an instruction i then
25                             Q_inst.push(i);
26                 else
27                     if op is an instruction i then
28                         Q_inst.push(i);
29     return S_v;
```

Source variable for definition. The definition is the address of store instruction. Similarly, we also use Algorithm 1 to extract the source variables of it. For instance, we assign the source variable of variable %25 in the *store* instruction as parameter %0 in Figure 4 for the `peek_token` function in Figure 1. For brevity, we omit the related codes in the `peek_token` function in Figure 4 since the tracking workflow is the same as the *br* instruction above.

Branch dependency construction. Overall, we use different strategies to match the source variables of definitions and usages with different types and instruction identifiers. ❶ For *global* or ❷ *local variables*, the matching is direct and sound since their source variables are globally or locally defined and the instruction identifiers are easily identified. ❸ Otherwise, if the type of source variable is *struct-typed pointer parameter*, we use a field-sensitive type-matching method to match the source variables based on whether their types of P_{st} are overlapped. For instance, the types of *a->b->c* and *b->c* are the matched but the the types of *b->c* and *b->d* are not. It's important to note that this matching is not completely accurate because there may be different instances of the matched type used by definitions and usages respectively. However, this

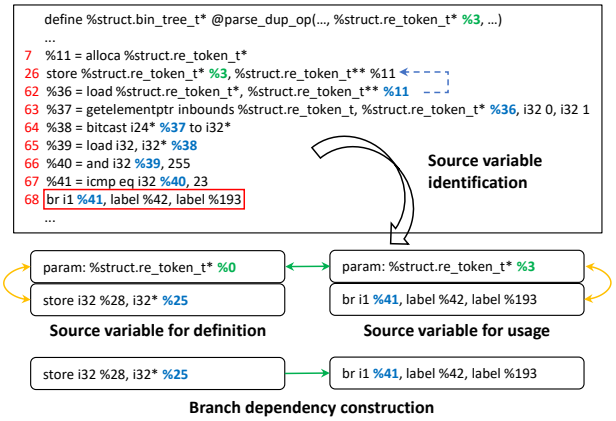

Figure 4: Workflow of branch dependency analysis.

over-approximation is typical in static analysis and is acceptable for higher completeness. For example, APICraft proposes a similar method called *Type-based transition* to statically build new inter-procedural data dependencies between library APIs [55]. ❹ For *non-pointer parameters*, we track them backward on call graph to locate their source variables in the caller functions. We then refine their source variables into the three types mentioned earlier for further matching. Finally, we build inter-procedural data dependency between the definitions and usages of the variables in branching conditions such as the *store* and *br* instructions in Figure 4.

3.2 Concrete Constraint Guided Searcher

This section presents our concrete constraint guided search strategy, which leverages the results of branch data dependency to select promising states for symbolic execution.

General symbolic execution. In general, the standard workflow of the symbolic execution technique is depicted in Algorithm 2, which is represented by black lines of code. In standard workflow, during each iteration of main loop, the symbolic execution engine chooses a state in line 6 based on the search strategy. If current state terminates or triggers an error, this state is removed and a new test case is generated if current path constraints are solvable in line 9-10. Otherwise, current instruction is symbolically interpreted based on its type. For branch instructions such as *br* or *switch* with a symbolic branching condition, new states are created via forking in line 13-14 after the SMT solver determines the solvability of the current path constraints. For other instructions, the current state advances one step within the current basic block in line 30. Finally, all states are updated for the next iteration in line 31.

Concrete constraint guided searcher. The goal of our searcher is to select promising states to satisfy the concrete constraints in uncovered concrete branches. Specifically, the selected state must satisfy the following two properties:

- It has covered the definitions of the variables in at least one partially covered concrete branching conditions.
- There existed a covered definition that can satisfy one of the constraint of uncovered concrete branch.

Algorithm 2: Workflow of symbolic execution

```
Input: Program(pgm), budget(bgt)
Output: A set of test cases(C)
1  Function SymExe(pgm, bgt):
2      S ← {pgm.initialState} ;                          ▷ set of states
3      C ← ∅ ;                                           ▷ set of test cases
4      V ← ∅ ;                                           ▷ set of metadata for concrete constraint
5      while bgt does not expire and (S ≠ ∅) do
6          state ← select(S);
7          inst ← state.inst;
8          if state exits or triggers error then
9              S ← S \ {state};
10             C ← C ∪ {solve(state)};
11         else if inst is a branch instruction β with condition φ then
12             if isSymbolic(φ) then
13                 Φ ← φ ∪ state.constraints;
14                 S ← S ∪ fork(state, Φ);
15             else
16                 store_β ← getStores(β);
17                 if β is partially covered then
18                     addStores(store_β);
19                     V ← V ∪ extractValidValue(φ);
20                     updateStates(S, V, store_β, β);
21                 else
22                     updateStates(S, ∅, ∅, β) ;
23                     removeStores(store_β);
24                 executeBrInst(β);
25         else if inst is a store instruction δ then
26             if isValidValue(V, δ.variable, δ.value) then
27                 updateStates(S, ∅, ∅, ∅);
28             executeStoreInst(δ);
29         else
30             state ← execute(inst);
31         S ← update(S);
32     return C;
```

To simplify the narration, we denote these definitions as *valid* definitions in the following paper. To this end, we propose a novel strategy to deal with each concrete constraint in a three-step cycle, which is shown in Algorithms 2 and 3. We add our methods in Algorithm 2 in blue lines of code, including the new handlers for branch instructions with concrete branching conditions in line 16-24, and new handler for store instructions in line 25-28. The metadata for each concrete constraint is recorded in a global map V, where each item consists of three elements: *<branch instruction, comparison operator, compared constant>*. In addition, Algorithm 3 mainly focuses on how to update states in each step.

Workflow of our searcher. Initially, we have extracted the definitions of the variables in branching conditions before symbolic execution in Section 3.1. During symbolic execution, our searcher leverages three steps to cyclically handle each concrete constraint in partly covered branching condition from when it is encountered for the first time (*Step 1*) to when it is fully covered (*Step 3*):

- *Step 1*: When a concrete branching condition is partially covered for the first time, we identify the unsatisfied concrete constraints to update states and start to track them.
- *Step 2*: We prioritize current state if it gives a *valid* definition to the variable in tracked concrete constraints.
- *Step 3*: We update states when a tracked concrete constraint is satisfied finally and stop to track it.

Note that *Step 2* is complementary to *Step 1* because it is likely that at the time of *Step 1*, some definitions are not covered by all current states. In addition, some prioritized states can not reach the concrete branching conditions. Next, we combine Algorithm 2 and Algorithm 3 to explain each step in details.

Step 1. When a concrete branching condition is encountered for the first time, we extract the store instructions associated with this branch instruction in line 16 and they are added to be globally tracked for *Step 2* in line 18 in Algorithm 2. Then, we use the `extractValidValue` function to analyze the covered concrete branching conditions to save the metadata in unsatisfied concrete constraints to V in line 19 by extracting the inverse comparison operator and the value of compared constant[1]. Next, in Algorithm 3, lines 3-5 iterate through all states to find those that have covered the store instructions mentioned earlier. For each of them, we extract the definition if it is constant and use the `isValidValue` function to determine whether current state satisfies the second property we list in this section in line 6-7. If so, we prioritize this state in next round in line 8 . Furthermore, we set this branch instruction as *target branch* and start to track it in line 9, which denotes the statement that this state steps towards.

Step 2. During symbolic execution, we track each store instruction added in *Step 1* for each state and record the written value only if it is constant. Since one state can globally assign different values to the same variable, we maintain a global mapping that links a variable to all the store instructions that define it and update its definition to the latest one. When we encounter the tracked store instructions, we also use the `isValidValue` function in line 26 in Algorithm 2 to determine whether the definition is *valid* as *Step 1* does. If so, we set this branch instruction to current state as *target branch* and prioritize current state in line 16-17 in Algorithm 3.

Step 3. Once current state reaches its *target branch*, it is highly likely that this uncovered concrete branch is covered. As a result, other states that are also directed towards this branch instruction are useless so their *target branches* are removed and and stopped to track in lines 11-13 of Algorithm 3. We also postpone the execution of a state if one state has no *target branches* in line 14 in Algorithm 3. Additionally, the *target stores* that were added in *Step 1* are removed from line 23 in Algorithm 2. To this end, we have completed the cycle for dealing with one concrete constraint in our searcher.

Except from this cycle, we classify all states into two sets based on whether they have *target branches* or not. When new states are generated, we always give priority to the states that have *target branches*. This strategy increases the possibility that these states will be able to fully cover new concrete branching conditions. In addition, for both sets of states, we employ a breadth-first search approach to state selection.

[1]For *switch* statements, we extract all of the values from uncovered cases.

Algorithm 3: State update for concrete branches

```
1  Function UpdateStates(S, V, stores, branch):
2      if branch is a new partially covered branch then
3          for state ∈ S do
4              δ_covered ← getCoveredStores(state);
5              δ_target ← δ_covered ∩ stores;
6              for δ ∈ δ_target do
7                  if isValidValue(V, δ.variable, δ.value) then
8                      prioritizeState(state);
9                      addTargetBranch(state, branch);
10     else if branch is a new fully covered branch then
11         for state ∈ S do
12             if branch ∈ state.targetBranches then
13                 removeTargetBranch(state, branch);
14                 delayState(state);
15     else
16         current ← getCurrent(S);
17         prioritizeState(current);
```

Example. We illustrate each step in Figure 5, which comprises one concrete branching condition *v>2* and four definitions of variable *v*. Step S0 is the initial state before symbolic execution and steps S1-S3 correspond to the three steps. ❶ In step S1, after one state has defined the variable *v* with value 1 and partially covers this concrete branching condition, we record the inverse comparison operator in satisfied concrete constraint as '>'and the constant value 2. Then we find all the covered definitions of *v* including *v=2* and *v=3* and prioritize the state that has covered definition *v=3* since this definition is *valid*. However, this state fails to reach the target branch perhaps due to other unsatisfied branching constraints. ❷ Then during step S2, when one state that has covered definition *v=2* covers definition *v=4*, we update the latest definition to *v=4* and prioritize this state since this definition is *valid*. ❸ Finally, in step S3, this state reaches and covers the uncovered concrete branch and the cycle to deal with this concrete constraint is completed.

3.3 Optimizations

According to our observations, it is possible that there are concrete constraints that are unable to be satisfied during symbolic execution. For example, some branching conditions are controlled by global macros, environment variables, configuration files and hardly triggered exceptions. In addition, there are probably some states that have covered valid definitions but can not reach their target branches. Therefore, since it can bring large overheads to track these target branches, we set a fixed number of the tracked target branches (e.g., 10) and periodically update them based on the number of executed instructions at runtime (e.g., 300000).

Specifically, we implement this mechanism by splitting tracked target branches into two FIFO queues. The first queue (Q1) is for tracking target branches and the size is fixed. The second queue (Q2) is varied-size and is filled when new concrete branching conditions are encountered only if the first queue is full. When the counter

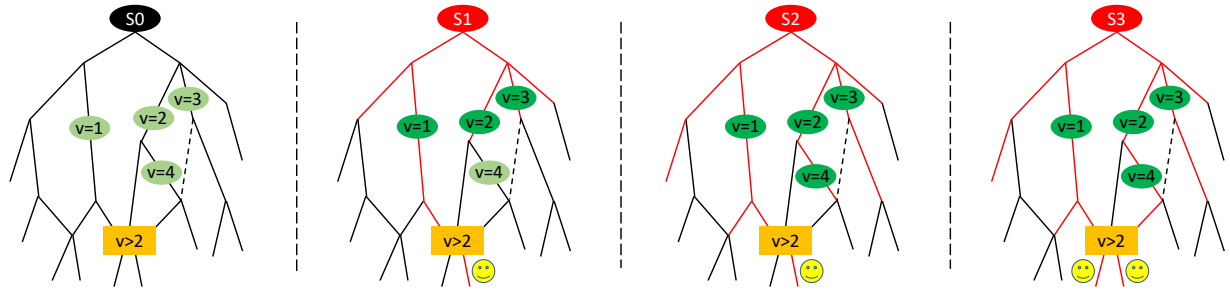

Figure 5: Workflow of concrete constraint guided searcher. The executed branches are in red color. The uncovered and covered definitions are in light and dark green color, respectively. The dash line denotes the branching constraint can not be satisfied.

of executed instructions expires during symbolic execution, Q1 is cleared and updated by Q2, which moves the first 10 new filled target branches to the Q1. Currently, these two hyper-parameters are unchanged during symbolic execution for all programs.

4 IMPLEMENTATION

We implement our design using about 3.2k LoC of C++ including 2k LoC for the branch dependency analysis module, and we add 1.2k Loc to KLEE 2.3 for our concrete constraint guided searcher on Ubuntu 18.04. We use the def-use chains, analysis passes for loops and control flow graph from LLVM 11.1.0. The added codes to KLEE mainly aim to analyse concrete constraints and manage states that partially or fully cover concrete branches.

Currently, our implementation only supports concrete constraints with integral comparisons. However, extending it to support float-point values and textual strings for different programs is straightforward. This can be achieved by adding new strategies to handle these types of values in the `extractValidValue` function in Algorithm 2. Furthermore, once we identify all the source variables of definitions and usages, we make a trade-off by focusing only on the branching conditions that only have a single variable in Section 3.1. The rationale behind this is that if a variable in a branching constraint has multiple source variables, its value cannot be determined by only checking the definition of one source variable's definition. This can result in incomplete coverage of target branches. Therefore, this trade-off ensures that the uncovered concrete branches are covered when a state, which has covered a specific definition, is reached. To support multiple source variables into our design, the main effort is to add a new handler in the `isValidValue` function in Algorithm 2 and 3. We further discuss this issue in depth in Section 7.

5 EVALUATION

In this section, we evaluate our methods mainly to answer the following four questions:

- **RQ1:** How many instructions are involved in the analysis of inter-procedural branch dependency?
- **RQ2:** Can our method improve branch coverage and also fully cover more concrete branching conditions ?
- **RQ3:** Can our optimization effectively and stably improve branch coverage?
- **RQ4:** Can our method trigger more security violations?

5.1 Experiment Setup

Benchmarks. We use 10 open-source C programs that are widely used by related work on search strategies for symbolic execution [6, 10, 12, 22]. The details on the benchmarks are listed in Table 1. For the harness to test libraries, we use an example from source codes for *expat*, and an example from official website from for *libxml2*. Since our methods hold the assumption that there are sufficient branching conditions that contain only one variable in the program, the size of the tested program should be larger enough to compare our method with other search strategies so that we do not use smaller standard programs such as *coreutils*.

Before running the programs, we use the methods in Section 3.1 to statically extract the branch dependency information and save it in newly generated programs in LLVM bitcode format, which are fed into symbolic execution tool for evaluation.

Baselines. Overall, all of the eleven search methods implemented in KLEE [7] are considered for our evaluation. The eight search methods we choose consists of `bfs` (breadth first search), `rss` (random state search), `rps` (random path search) and five instances of the nurs (non uniform random search) family including `nurs:rp` (nurs with 1/2^depth), `nurs:covnew`, `nurs:cpicnt`, `nurs:md2u` and `nurs:qc`. We do not use `dfs` (depth first search) and `nurs:depth` since they are similar with `nurs:rp` and `nurs:rp` performs best on the average coverage for most programs. In addition, `nurs:cpicnt` and `nurs:icnt` both use the guidance of the number of instructions so we reserve the better one `nurs:cpicnt`. We denote our searcher as `cgs` (concrete-constraint guided searcher). Therefore, in total, there are eight search methods in our baselines and we adopt each of them in each program for evaluation.

Settings. All experiments are conducted on a Linux machine with 4 Intel(R) Gold(R) 5218 CPUs (2.30GHz) and 128GB RAM, where it has a total of 16 cores and 64 threads. We test each program for 8 hours for 8 times with 4G memory limit. We use the default definitions for symbolic inputs from KLEE in the last column of Table 1. In addition, we set the *–optimize* option of KLEE to *false* since it is likely that the extracted branch dependency information is removed before symbolic execution.

Table 1: The details of the benchmarks used for evaluations. Branches are collected using KLEE internal coverage. The KLEE format of symbolic inputs are set based on the common usage of programs.

Program	Version	Binary Size	Branches	Lines	KLEE formats of symbolic inputs
expat	2.5.0	390KB	3568	6919	–sym-stdin 100
libxml2	2.10.3	2.5MB	30414	70153	A –sym-files 1 40
readelf	2.36	2.5MB	10942	38333	-a A –sym-files 1 100
objcopy	2.36	5.1MB	22393	56668	–sym-args 0 3 20 A –sym-files 1 100
gawk	5.1.0	1.4MB	10041	25553	-f A B –sym-files 2 50
make	4.3	523KB	4874	8490	-n -f A –sym-files 1 40
sqlite3	3.39.4	2.5MB	16686	48966	–sym-stdin 20 –sym-stdout
nasm	2.15.05	2.3M	6327	18827	–sym-args 0 2 2 A –sym-files 1 100
grep	3.6	632kB	5110	10879	–sym-arg 10 –sym-args 0 2 2 –sym-files 1 8 –sym-stdin 8
sed	4.8	508KB	4338	9258	–sym-arg 10 –sym-args 0 2 2 –sym-files 1 8 –sym-stdin 8

5.2 Branch Dependency

This section shows the results of branch dependency analysis including the number of involved instructions and the number of different types of source variables of involved branches. The branches and stores involved in this module are used by our concrete constraint guided searcher for state selection.

In general, the total number of source variables is greater than the number of branch instructions because some variables in branching conditions share the same source variables and not all branch instructions can match store instructions. On average, The branches account for about 15% ∼ 20% of all branches by comparing data in Table 1 and Table 2. This proportion seems relatively small because we only focus on the variables that have only one source variable in branching conditions. However, as we explain in Section 4, this trace-off guarantees that the concrete constraints can be satisfied when one state that has covered specific definition reaches. Therefore, our methods strictly limit the branching conditions used for our concrete constraint guided searcher.

Table 2: Statistics of branch dependency analysis for 10 real-world programs. The branch and store instructions are the usage and definition sites of inter-procedural data dependency, respectively. For the types of source variables of branches, GV=*Global Variable*, LV=*Local Variable*, NPP=*Non-Pointer Parameter*, SPP=*Struct-typed Pointer Parameter*.

Program	Instructions		Types of source variables			
	Branch	Store	GV	LV	NPP	SPP
expat	388	951	0	232	125	174
libxml2	3585	4771	163	1298	1583	3976
readelf	2321	3085	350	805	1227	1113
objcopy	3858	5544	155	1578	762	3751
gawk	1992	2072	599	422	737	952
make	582	830	243	255	129	136
sqlite3	3797	5484	21	1405	846	3139
nasm	770	1142	116	273	530	476
grep	1029	1270	64	318	448	667
sed	768	1100	36	266	423	487

* The branches also contain *switch* instructions.

5.3 Branch Coverage

In this section, we evaluate our search strategy by examining the branch coverage improvements on KLEE [7]. To start, we present the total branch coverage over time, showcasing the mean values and 95% standard deviations for each program and search strategy. Next, we report the final number of fully covered concrete branching conditions. This analysis helps us to shed light on the reason of the improvements on overall branch coverage by our concrete constraint guided searcher.

We collect the results from KLEE's statistics during execution and show them in Figure 6 for each tested program. Overall, the experimental results show that `cgs` achieves the highest branch coverage for 9 programs and the second best for *nasm*. After 8h time budget expires, the average branch coverage of `cgs` is 2243.5, which outperforms the second ranked strategy `bfs` with 1741.5 by 28.8% and the third ranked strategy `nurs:rp` with 1518.0 by 47.8%. In particular, for *grep*, *gawk*, *expat*, *readelf* and *objcopy*, `cgs` notably increases the covered branches than the second best strategy. For instance, `cgs` notably outperforms the second best strategy *bfs* for *objcopy* by 1360 branches on average. Moreover, as shown in *readelf* and *objcopy*, the rate for the branch coverage increase over time is noticeably higher than other strategies.

For *nasm*, `nurs:rp` is the best strategy and it finally covers 48 branches more than `cgs`. The standard deviations for *sqlite3* is huge for `bfs` and `cgs` since there are several branches control a large number branches and they both can not cover these branches in some runs. In addition, we find that `random-state`, `nurs:covnew`, `nurs:cpicnt` and `nurs:md2u` always stop executing to solve branching constrains, which spends so little time for instruction execution that the final branch coverage and standard deviations are both rather small.

Concrete branching conditions. Furthermore, to demonstrate whether `cgs` improves branch coverage by fully covering more concrete branching conditions, we collect the final number of them with each strategy for each program after 8h. We only list the mean values of the number of these branches in Table 3. Finally, `cgs` fully covers 46.6 concrete branching conditions on average that we instrument before symbolic execution compared to `bfs` with 31.0, which improves 15.6 concrete branching conditions. In

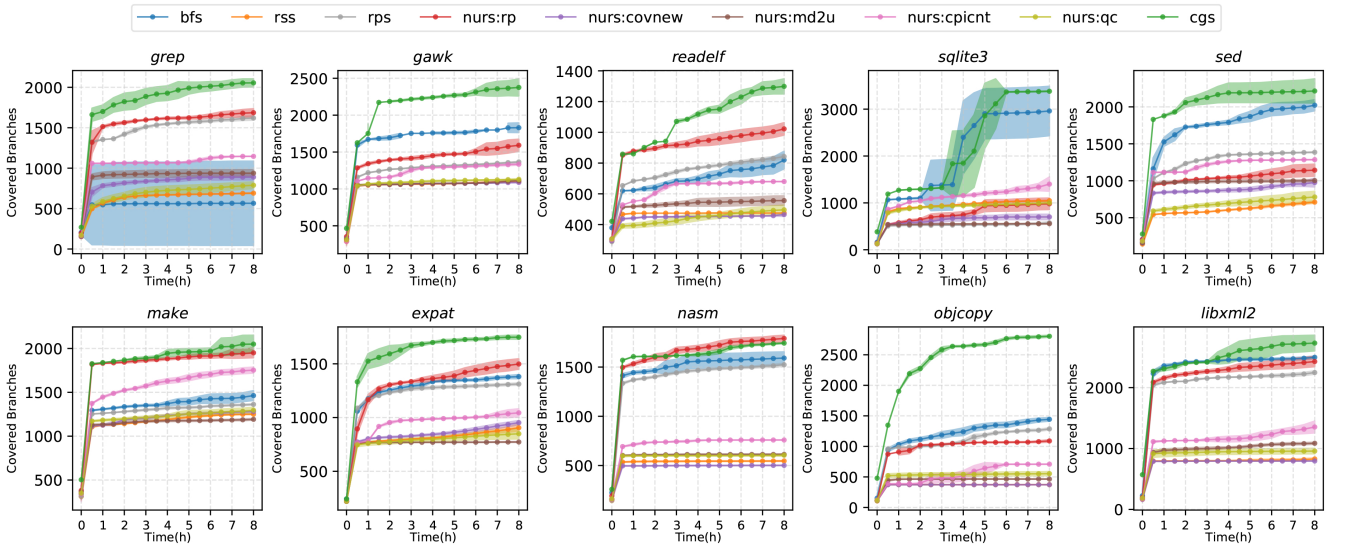

Figure 6: Branch coverage for 10 real-world programs by running KLEE with different strategies for 8h. Mean values and 95% standard deviations over 8 runs are shown in solid lines and shadows, respectively.

particular, for *grep*, *gawk*, *expat*, *sed* and *objcopy*, `cgs` increases the number of fully covered concrete branching conditions with a rather significant advantage over the second best strategies. However, for *readelf* and *make*, `nur:cpicnt` and `nurs:rp` perform better than `cgs` with a rather small gap for 2.8 and 1.3, respectively.

In general, there are two main reasons why overall branch coverage can be improved when more covered concrete branching conditions are fully covered. Firstly, these concrete constraints govern new sections of code that include numerous new branches, both symbolic and concrete. Secondly, new definitions for variables in branching constraints introduce more covered branches as the new branch constraints are satisfied. In summary, the findings presented in Table 3 provide evidence of the effectiveness of our method and offer an explanation for the significant improvement in branch coverage.

Table 3: The number of fully covered concrete branching conditions for 10 real-world programs by running KLEE with different strategies for 8h. Only mean values are listed.

	bfs	rss	rps	nurs: rp	nurs: covnew	nurs: cpicnt	nurs: md2u	nurs: qc	cgs
expat	17.8	13.2	15.9	20.9	15.0	15.0	11.6	11.9	**36.6**
libxml2	31.1	5.4	21.2	29.0	5.0	10.0	7.0	7.8	**41.8**
readelf	3.1	0.1	4.0	**13.2**	1.0	4.0	1.4	0.0	10.4
objcopy	13.8	4.0	8.0	4.2	4.0	4.0	4.0	4.0	**33.5**
gawk	41.2	14.0	22.9	30.6	13.9	16.5	13.8	16.2	**61.6**
make	26.1	25.2	25.8	30.2	26.1	**32.4**	22.0	25.0	31.1
sqlite3	84.0	7.8	4.2	8.8	3.6	19.5	6.4	7.6	**130.0**
nasm	20.8	7.0	17.1	**23.0**	6.0	9.0	6.0	6.9	22.4
grep	25.0	6.0	31.5	33.6	10.6	14.0	11.0	6.1	**63.5**
sed	47.4	3.9	7.4	3.0	6.1	11.0	3.1	3.2	**62.2**
average	31.0	8.7	15.8	20.0	9.6	13.5	8.6	8.9	**46.6**

5.4 Hyper-parameters

In this section, we evaluate the effectiveness and stableness of our optimization in Section 3.3 by setting two hyper-parameters including the fixed number of the tracked target branches (i.e., N_{br}) and the number of executed instructions (i.e., N_{inst}) to update these branches. Note that we both evaluate whether this setting can improve branch coverage and the sensitivity of results with different pairs of two hyper-parameters. Specifically, for the first goal, we set (N_{br}, N_{inst}) to (0xffff, 0xffffffff) to track all target branches and do not update them so the optimization is off. For the second we use 4x4 combinations of two hyper-parameters including [5, 10, 30, 100] for N_{br} and [10000, 100000, 300000, 1000000] for N_{inst} to show the sensitivity of results with different pairs of them. We run each setting three times on all programs for 8h and collect the average branch coverage to show the results.

In Figure 7, the y-axis shows the ratio of the average branch coverage with optimization to the average branch coverage without optimization, which is denoted as $BCov_o/BCov_{wo}$. In addition, the value of N_{inst} is shown on x-axis and the value of N_{br} is shown at

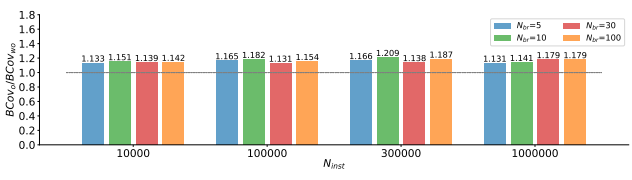

Figure 7: Branch coverage improvement with optimizations by setting different pairs of hyper-parameters.

Table 4: UBSan Errors triggered by the test cases generated using the top-4 strategies on part of programs. UIO=Unsigned Integer Overflow, NPD=Null Pointer Dereference.

Program	Type	Error Location	cgs	bfs	nurs:rp	rps
gawk	UIO	L.284 in re.c	✓	✓		✓
grep	UIO	L.132 in grep.c	✓		✓	✓
objcopy	UIO	L.233 in elfcore.h	✓			
	UIO	L.544 in tekhex.c	✓			
sed	NPD	L.1221 in compile.c	✓			
	NPD	L.1281 in execute.c			✓	✓
make	UIO	L.2354 in read.c	✓	✓	✓	

the upper right of Figure 7. The $BCov_{wo}$ reaches 1900 after running for 8h. In brief, the values of $BCov_o/BCov_{wo}$ are greater than 1.0 on all pairs of two hyper-parameters so that our optimization is proved effective to improve branch coverage. Moreover, the improvements with all settings on two hyper-parameters are nearly around 15% and very close to each other. Therefore, our optimization is stable with a low sensitivity on the improvement of branch coverage.

5.5 Security Violations

In this section, we replay the test cases generated by all strategies on the program instrumented with UBSan (Undefined Behavior Sanitizer) checkers to evaluate their capability of detecting security violations [2]. We only focus on seven types of UBSan checks including signed and unsigned integer overflow, shift overflow, out of bounds array indexing, pointer-overflow and null pointer deference because these bugs are relatively common in C programs [20].

We replay the test cases generated from all runs for each program and use the best one as the results. If the UBSan errors can be triggered by all of the four strategies, we do not list them. Table 4 shows the program, the type of UBSan errors, the error-location and whether the error can be triggered using the test cases generated after 8h by each strategy, which is marked with a "✓" if success. Overall, for the 7 triggered errors from five programs, `cgs` successes for six of them except one for *sed* while others success for at most three of them. In particular, `cgs` finds both UBSan errors in *objcopy*, which is reasonable since `cgs` also achieves a novel increase on branch coverage. In short, due to the ability of `cgs` on path exploration, the symbolic execution engine can find more security violations with the same time budget.

6 RELATED WORK

Symbolic execution techniques have attracted much attention in recent years in many aspects such as path explosion [10–12, 22, 35, 50, 52, 53], execution engine [15, 33, 34, 54], constraint solving [3, 38, 44, 46, 51], memory model [41, 42, 49] and so on. In this paper, we summarize related work to ours mainly in two aspects.

Search strategies for symbolic execution. To solve the *path explosion* program, various search strategies with different goals, characteristics and capabilities have been proposed for symbolic execution tools to select the most suitable candidate state. HOMI ranks states using the covered branches that contribute to determining the value of a test case, which can effectively increase branch coverage [12]. DiSE leverages static analysis techniques to compute program difference information to explore program execution paths and generate path conditions affected by the differences [32]. To improve the efficiency for bug reproductions, the error traces of static analysis reports are used to guide symbolic execution tool [6]. Ferry focuses on program-state-dependent branches and guide symbolic execution using the behaviours of state variables [56]. Our work spends more efforts on how to fully cover more concrete branching conditions to improve overall code coverage.

Data dependency guided software testing. According to lines of research, the data dependency information represents a suitable candidate for program state descriptions during software testing. DDFuzz uses data dependency graphs as new code coverage metrics to guide existed fuzzing tools such as AFL++ [16, 31]. DGSE relies on the analysis of data dependencies to determine whether different visits to a statement instance produce the same symbolic value for symbolic execution [43]. GREYONE proposes a fuzzing-driven taint inference to identify taint of variables, which are used to mutate input bytes during fuzzing [18]. PATA identifies the variables used in branching constraints and constructs the representative variable sequence for input mutation in fuzzing [30]. Instead, our work prioritizes states during symbolic execution based on reliable inter-procedural data dependency on the variables used in concrete constraints to increase overall code coverage.

7 FUTURE WORK

In this paper, we introduce a new issue in our design regarding the handling of multiple source variables for a single variable in one branching constraint. Generally, when a branching constraint's outcome depends on more than one source variable, it becomes unreliable to prioritize a single state when it defines only one variable. To tackle this issue, we can use an alternative approach: synthesizing a symbolic function that connects the branching constraint variable with all of its source variables before symbolic execution. These synthesized symbolic functions are then loaded before symbolic execution. When the involved branches are encountered, the variables are concretized to calculate the results of the variable in the branching constraint, which are subsequently utilized by our search strategy for path prioritization.

8 CONCLUSION

In this work, we introduce Concrete Constraint Guided Symbolic Execution, a novel approach to effectively improve overall code coverage. The core idea of our approach involves extracting concrete constraints from partially covered concrete branching conditions and leveraging inter-procedural data dependency to improve symbolic execution. By investigating whether a state has carried variable definitions that satisfy the constraints of uncovered concrete branch conditions, we can achieve better overall branch coverage. The experimental results demonstrate the effectiveness of our method on increasing overall code coverage and the number of security violations during symbolic execution on 10 real-world C programs.

9 ACKNOWLEDGEMENT

This work is supported by National Key R&D Program of Ministry of Science and Technology (No.2021YFB3101803), Projects from the Ministry of Industry and Information Technology of China (No. TC220H055).

REFERENCES

[1] 2023. *Source code of grep 3.6.* https://ftp.gnu.org/gnu/grep/grep-3.6.tar.gz
[2] 2023. *UndefinedBehaviorSanitizer - Clang 17.0.0git documentation.* https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
[3] Alireza S. Abyaneh and Christoph M. Kirsch. 2021. ASE: A Value Set Decision Procedure for Symbolic Execution. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 203–214. https://doi.org/10.1109/ASE51524.2021.9678584
[4] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 367–381.
[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (may 2018), 39 pages. https://doi.org/10.1145/3182657
[6] Frank Busse, Pritam Gharat, Cristian Cadar, and Alastair F. Donaldson. 2022. Combining Static Analysis Error Traces with Dynamic Symbolic Execution (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 568–579. https://doi.org/10.1145/3533767.3534384
[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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* (San Diego, California) *(OSDI'08)*. USA, 209–224.
[8] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (feb 2013), 82–90. https://doi.org/10.1145/2408776.2408795
[9] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-Agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 746–759. https://doi.org/10.1145/3427228.3427280
[10] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1244–1254. https://doi.org/10.1145/3180155.3180166
[11] Sooyoung Cha and Hakjoo Oh. 2019. Concolic Testing with Adaptively Changing Search Heuristics *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 235–245. https://doi.org/10.1145/3338906.3338964
[12] Sooyoung Cha and Hakjoo Oh. 2020. Making Symbolic Execution Promising by Learning Aggressive State-Pruning Strategy *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 147–158. https://doi.org/10.1145/3368089.3409755
[13] Vitaly Chipounov. 2022. *The S2E Platform: A Journey from a Research Prototype to a Commercial Product.* https://srg.doc.ic.ac.uk/klee22/talks/Chipounov-S2E-Platform.pdf
[14] Lori A. Clarke. 1976. A Program Testing System. In *Proceedings of the 1976 Annual Conference* (Houston, Texas, USA) *(ACM '76)*. Association for Computing Machinery, New York, NY, USA, 488–491. https://doi.org/10.1145/800191.805647
[15] Emilio Coppa, Heng Yin, and Camil Demetrescu. 2022. SymFusion: Hybrid Instrumentation for Concolic Execution. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 100, 12 pages. https://doi.org/10.1145/3551349.3556928
[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies (WOOT'20)*. USENIX Association, USA, Article 10, 1 pages.
[17] Farhaan Fowze, Dave Tian, Grant Hernandez, Kevin Butler, and Tuba Yavuz. 2021. ProXray: Protocol Model Learning and Guided Firmware Analysis. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1907–1928. https://doi.org/10.1109/TSE.2019.2939526
[18] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 145, 18 pages.
[19] Indradeep Ghosh. 2018. *Utilization and Evolution of KLEE-based Technologies for Embedded Software Testing at Fujitsu.* https://srg.doc.ic.ac.uk/klee18/talks/Ghosh-Keynote.pdf
[20] Satyajit Ghosh. 2022. *Security issues in C language.* https://www.geeksforgeeks.org/security-issues-in-c-language/
[21] Peter Goodman. 2022. *Can Symbolic Execution Be a Productivity Multiplier for Human Bug-Finders?* https://srg.doc.ic.ac.uk/klee22/talks/Goodman-SE-for-Bug-Finders.pdf
[22] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to Explore Paths for Symbolic Execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2526–2540. https://doi.org/10.1145/3460120.3484813
[23] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin R.B. Butler. 2017. FirmUSB: Vetting USB Device Firmware Using Domain Informed Symbolic Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2245–2262. https://doi.org/10.1145/3133956.3134050
[24] Evan Johnson, Maxwell Bland, Yi Fei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted firmware rehosting for embedded systems. In *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, 321–338.
[25] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (jul 1976), 385–394. https://doi.org/10.1145/360248.360252
[26] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 438–450. https://doi.org/10.1145/3510003.3510628
[27] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 193–204. https://doi.org/10.1145/2254064.2254088
[28] Peng Li, Rundong Zhou, Yaohui Chen, Yulong Zhang, and Tao (Lenx) Wei. 2018. *ConcFuzzer: A Sanitizer Guided Hybrid Fuzzing Framework Leveraging Greybox Fuzzing and Concolic Execution,.* https://srg.doc.ic.ac.uk/klee18/talks/Li-Keynote.pdf
[29] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 19–32. https://doi.org/10.1145/2509136.2509553
[30] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1–17. https://doi.org/10.1109/SP46214.2022.9833594
[31] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with Data Dependency Information. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. 286–302. https://doi.org/10.1109/EuroSP53844.2022.00026
[32] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed Incremental Symbolic Execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 504–515. https://doi.org/10.1145/1993498.1993558
[33] Sebastian Poeplau and AurÃ©lien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA, 181–198.
[34] Sebastian Poeplau and AurÃ©lien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *Network and Distributed System Security Symposium*. Network & Distributed System Security Symposium.
[35] Rui Qiu, Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. 2015. Compositional Symbolic Execution with Memoized Replay. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 632–642. https://doi.org/10.1109/ICSE.2015.79
[36] Eric F. Rizzi, Sebastian Elbaum, and Matthew B. Dwyer. 2016. On the Techniques We Create, the Tools We Build, and Their Misalignments: A Study of KLEE. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 132–143. https://doi.org/10.1145/2884781.2884835
[37] Nicola Ruaro, Kyle Zeng, Lukas Dresel, Mario Polino, Tiffany Bao, Andrea Continella, Stefano Zanero, Christopher Kruegel, and Giovanni Vigna. 2021. SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID '21)*. Association for Computing Machinery, New York, NY, USA, 456–468. https://doi.org/10.1145/3471621.3471865

[38] Ziqi Shuai, Zhenbang Chen, Yufeng Zhang, Jun Sun, and Ji Wang. 2021. Type and interval aware array constraint solving for symbolic execution. *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2021).

[39] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

[40] Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. 2022. Symbolic Execution for Randomized Programs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 181 (oct 2022), 30 pages. https://doi.org/10.1145/3563344

[41] David Trabish, Shachar Itzhaky, and Noam Rinetzky. 2021. A Bounded Symbolic-Size Model for Symbolic Execution. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1190–1201. https://doi.org/10.1145/3468264.3468596

[42] David Trabish and Noam Rinetzky. 2020. Relocatable Addressing Model for Symbolic Execution. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/3395363.3397363

[43] Haijun Wang, Ting Liu, Xiaohong Guan, Chao Shen, Qinghua Zheng, and Zijiang Yang. 2017. Dependence Guided Symbolic Execution. *IEEE Trans. Softw. Eng.* 43, 3 (mar 2017), 252–271. https://doi.org/10.1109/TSE.2016.2584063

[44] Junye Wen, Tarek Mahmud, Meiru Che, Yan Yan, and Guowei Yang. 2023. Intelligent Constraint Classification for Symbolic Execution. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 144–154. https://doi.org/10.1109/SANER56733.2023.00023

[45] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang. 2022. Evaluating and Improving Neural Program-Smoothing-Based Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 847–858. https://doi.org/10.1145/3510003.3510089

[46] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. 2021. Boosting SMT Solver Performance on Mixed-Bitwise-Arithmetic Expressions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 651–664. https://doi.org/10.1145/3453483.3454068

[47] Carter Yagemann, Simon P. Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 320–336. https://doi.org/10.1145/3460120.3485363

[48] Carter Yagemann, Matthew Pruett, Simon Pak Ho Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *USENIX Security Symposium*.

[49] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 144–154. https://doi.org/10.1145/2338965.2336771

[50] Guowei Yang, Rui Qiu, Sarfraz Khurshid, Corina S. Păsăreanu, and Junye Wen. 2019. A synergistic approach to improving symbolic execution using test ranges. *Innovations in Systems and Software Engineering* 15, 3, 325–342. https://doi.org/10.1007/s11334-019-00331-9

[51] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast Bit-Vector Satisfiability. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 38–50. https://doi.org/10.1145/3395363.3397378

[52] Qiuping Yi and Guowei Yang. 2022. Feedback-Driven Incremental Symbolic Execution. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. 505–516. https://doi.org/10.1109/ISSRE55969.2022.00055

[53] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2015. Postconditioned Symbolic Execution. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10. https://doi.org/10.1109/ICST.2015.7102601

[54] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 745–761.

[55] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2811–2828.

[56] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. 2022. Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4365–4382.

[57] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association.