Combining Symbolic Execution and Model Checking for Data Flow Testing

Ting Su† Zhoulai Fu‡ Geguang Pu† Jifeng He∗ Zhendong Su†

∗Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
†Department of Computer Science, University of California, Davis, USA
Email: tsuletgo@gmail.com, zlfu@ucdavis.edu,
ggpu@sei.ecnu.edu.cn (†corresponding author), jifeng@sei.ecnu.edu.cn, su@cs.ucdavis.edu

Abstract—Data flow testing (DFT) focuses on the flow of data through a program. Despite its higher fault-detection ability over other structural testing techniques, practical DFT remains a significant challenge. This paper tackles this challenge by introducing a hybrid DFT framework: (1) The core of our framework is based on dynamic symbolic execution (DSE), enhanced with a novel guided path search to improve testing performance; and (2) we systematically cast the DFT problem as reachability checking in software model checking to complement our DSE-based approach, yielding a practical hybrid DFT technique that combines the two approaches’ respective strengths. Evaluated on both open source and industrial programs, our DSE-based approach improves DFT performance by 60–80% in terms of testing time compared with state-of-the-art search strategies, while our combined technique further reduces 40% testing time and improves data-flow coverage by 20% by eliminating infeasible test objectives. This combined approach also enables the cross-checking of each component for reliable and robust testing results.

I. INTRODUCTION

Testing is the most widely adopted software validation technique. Structural coverage criteria, such as statement, branch and logical [1]–[3], have been widely used to assess test adequacy. In contrast to these structural criteria, data flow criteria [4]–[7] focus on the flow of data through a program, i.e. the interactions between variable definitions and their corresponding uses. The motivation is to verify the correctness of defined variable values by observing that all corresponding uses of these values produce the desired results.

According to several empirical studies [1], [8], [9], data-flow criteria are more effective than control flow-based testing criteria (e.g. statement or branch). However, several reasons hinder the adoption of data flow testing in practice. First, few data flow coverage tools exist. To our knowledge, ATAC [10], [11] is the only available tool to compute data flow coverage for C programs developed two decades ago. Second, the complexity of identifying data flow-based test data [12], [13] overwhelms software testers: test objectives w.r.t. data-flow criteria are much more than those of structural criteria; more efforts are required to derive a test case to cover a variable definition and its corresponding use than just covering a statement or branch. Third, infeasible test objectives (i.e. paths from definitions to their uses are infeasible) and variable aliases make data flow testing more difficult.

The aforementioned challenges underline the importance of effective automated data flow testing. To this end, this paper presents a combined approach to automatically generate data flow-based test data. Our approach synergistically combines two techniques: dynamic symbolic execution and counterexample-guided abstraction refinement-based model checking. At the high level, given a program as input, our approach (1) outputs test data for feasible test objectives and (2) eliminates infeasible test objectives — without any false positives.

Dynamic Symbolic Execution [14], [15] (DSE) is a widely accepted and effective approach for automatic test data generation. It intertwines traditional symbolic execution [16] with concrete execution, and explores as many program paths as possible to generate test cases by solving path constraints. As for Counterexample-Guided Abstraction Refinement-based (CEGAR) model checking [17]–[19], given the program source and a temporal safety specification, it either statically proves that the program satisfies the specification or produces a counterexample path that demonstrates the violation. It has been applied to automatically verify safety properties of OS device drivers [17], [20], [21] and generate test data w.r.t. statement or branch coverage [22] from counterexample paths.

Although DSE has been widely adopted to achieve different coverage criteria (e.g. branch, logical, boundary value and mutation testing [23]–[27]), little effort exists to adapt DSE to data flow testing. To mitigate path explosion in symbolic execution, we design a guided path search strategy to cover data-flow test objectives as quickly as possible. With the help of concrete executions in DSE, we can also more easily and precisely detect definitions due to variable aliasing. Moreover, we introduce a simple, powerful encoding of data flow testing using CEGAR-based model checking to complement our DSE-based approach: (1) We show how to encode any data-flow test objective in the program under test and systematically evaluate the technique’s practicality; and (2) we describe a combined approach that combines the relative strengths of the DSE and CEGAR-based approaches. An interesting by-product of this combination is to let the two independent approaches cross-check each other’s results for correctness and consistency.

We have implemented our data flow testing framework and the guided path search strategy on top of a DSE engine named CAUT, which has been continuously developed and refined in previous work [25], [26], [30]. We perform data flow testing on four open source and two industrial programs in C. By comparing the performance of our proposed search strategy against other popular search strategies [31], [32], our DSE-based approach can improve data flow testing by 60–80% in terms of testing time. In addition, we have adapted the CEGAR-based approach to complement the DSE-based approach. Evaluation results show that it can reduce testing time by 40% than using the CEGAR-based approach alone and improve coverage by 20% than using the DSE-based approach alone. Thus, indeed
our combined approach provides a more practical means for data flow testing.

In summary, we make the following main contributions:

- We design a DSE-based data flow testing framework and enhance it with an efficient guided path search strategy to quickly achieve data-flow coverage criteria. To our knowledge, our work is the first to adapt DSE for data flow testing.

- We describe a simple, effective reduction of data flow testing to reachability checking in software model checking [20], [22] to complement our DSE-based approach. Again to our knowledge, we are the first to systematically adapt CEGAR-based approach to aid data flow testing.

- We realize the DSE-based data flow testing approach and conduct empirical evaluations on both open source and industrial C programs. Our results show that the DSE-based approach is both efficient and effective.

- We also demonstrate that the CEGAR-based approach effectively complements the DSE-based approach by further reducing data flow testing time and detecting infeasible test objectives. In addition, these two approaches can cross-check each other to validate the correctness and effectiveness of both techniques.

The rest of the paper is organized as follows. Section II introduces necessary background and gives an overview of our data flow testing approach. Section III details our DSE-based approach and our reduction of data flow testing to reachability checking in model checking. Next, we present details of our implementation (Section IV) and empirical evaluation (Section V). Section VI surveys related work, and Section VII concludes.

II. OVERVIEW

A. Problem Setting

A program path is a sequence of control points (denoted by line numbers), written in the form $l_1, l_2, \ldots, l_n$. We distinguish two kinds of paths. A control flow path is a sequence of control points along the control flow graph of a program; an execution path is a sequence of executed control points driven by a program input.

Following the classic definition from Herman [4], a def-use pair $du(l_d, l_u, x)$ occurs when there exists at least one control flow path from the assignment (i.e. definition, or def in short) of variable $x$ at control point $l_d$ to the statement at control point $l_u$ where the same variable $x$ is used (i.e. use) on which no redefinitions of $x$ appear (i.e. the path is def-clear).

Definition 1 (Data Flow Testing): Given a def-use pair $du(l_d, l_u, x)$ in program $P$, the goal of data flow testing is to find an input $t$ that induces an execution path that covers (i.e., passes through) $l_d$ and then $l_u$ with no intermediate redefinitions of $x$ between $l_d$ and $l_u$. The requirement to cover all def-use pairs at least once is called all def-use coverage criterion.

In this paper, we use dynamic symbolic execution (DSE) [14], [15] to generate test inputs to satisfy def-use pairs. The DSE-based approach starts with an execution path triggered by an initial test input and then iterates the following: from an execution path $p = l_1, \ldots, l_{i-1}, l_i, \ldots, l_n$, DSE picks an executed branch (i.e. a branching node) of a conditional statement at $l_i$ (the choice depends on an underlying search strategy). It then solves the path constraint collected along $l_1, \ldots, l_{i-1}$ conjuncted with the negation of the executed branch condition at $l_i$ to find a new test input. This input will be used as a new test case in the next iteration to generate a new path $p' = l_1, \ldots, l_{i-1}, l_i, \ldots$, which deviates from the original path $p$ at $l_i$ (the opposite branch direction of the original executed branch at $l_i$), but shares the same path prefix $l_1, \ldots, l_{i-1}$ with $p$. If the target def-use pair is covered by this new path $p'$ (cf. Definition 1), we obtain the test case which satisfies this pair. Otherwise, the process will continue until a termination condition (e.g. a time bound is reached or the whole path space has been explored) is met.

Although the DSE-based technique is an effective way to generate test inputs to cover specified program points, it faces two challenges when applied in data flow testing:

1) The DSE-based technique by nature faces the notorious path-explosion problem. It is challenging, in reasonable time, to find an execution path from the whole path space to cover a target pair.

2) The test objectives from data flow testing include feasible and infeasible pairs. A pair is feasible if there exists an execution path which can pass through it. Otherwise it is infeasible. Without prior knowledge about whether a target pair is feasible or not, DSE-based approach may spend a large amount of time, in vain, to cover an infeasible def-use pair.

B. An Illustrative Example

We give an overview of our approach via a simple program power (Figure 1), which takes as input two integers $x$ and $y$ and outputs $x^y$. Its control flow graph is shown in the right column in Figure 1. For illustration, we will explain how our approach deals with the aforementioned challenges demonstrated by the

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{power.png}
\caption{An example: power.}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Column 1} & \textbf{Column 2} \\
\hline
1 & double power(int x, int y){
2 & \hspace{1cm} int exp;
3 & \hspace{1cm} double res;
4 & \hspace{1cm} if (y>0)
5 & \hspace{2cm} exp = y;
6 & \hspace{1cm} else
7 & \hspace{2cm} exp = -y;
8 & \hspace{1cm} res=1;
9 & \hspace{1cm} while (exp!=0){
10 & \hspace{2cm} res *= x;
11 & \hspace{2cm} exp -= 1;
12 & \hspace{2cm} }
13 & \hspace{1cm} if (y<=0)
14 & \hspace{2cm} res *= x;
15 & \hspace{1cm} return res;
16 & \hspace{1cm} abort;
17 & \hspace{1cm} return 1.0/res;
18 & \hspace{1cm} return res;
19 & }
\hline
\end{tabular}
\end{table}

\footnote{A branching node is an execution instance of an original branch in the code. When a conditional statement is inside a loop, it can correspond to multiple branching nodes along an execution path.}
Following two pairs w.r.t. the variable res:
\[
\begin{align*}
du_1 &= (l_9, l_{17}, res) \\
ru_2 &= (l_9, l_{18}, res)
\end{align*}
\] (1)
(2)

We combine DSE to quickly cover \(du_1\) and CEGAR to prove \(du_2\) is infeasible. The details of these two approaches are explained in Section III.

**DSE-based Data Flow Testing** DSE starts by taking an arbitrary test input \(t\), e.g., \(t = (x \rightarrow 0, y \rightarrow 42)\). This test input triggers an execution path \(p\)
\[
p = l_4, l_5, l_8, l_9, l_{10}, l_{11}, l_9, l_{10}, l_{11}, \ldots, l_9, l_{13}, l_{18}
\]
repeated 42 times (3)

which covers the def of \(du_1\) at \(l_8\). To cover its use, the classical DSE approach (e.g. with depth-first or random path search [32]) will systematically flip branching nodes on \(p\) to explore new paths until the use is covered. However, the problem of path explosion — hundreds of branching nodes on path \(p\) (including nodes from new paths generated from \(p\)) can be flipped to fork new paths — could greatly slow down the exploration. We use two techniques to tackle this problem.

First, we use the redefinition pruning technique to remove invalid branching nodes: res is detected as redefined on \(p\) at \(l_{10}\) in dynamic execution, so it is useless to flip the branching nodes after the redefinition point (the paths passing through the redefinition point cannot satisfy the pair). To illustrate, we cross out these invalid branching nodes on \(p\) and highlight the rest in Fig. 2. As we can see, a large number of invalid branching nodes can be pruned.

\[
p = l_4, l_5, l_8, l_9, l_{10}, l_{11}, l_9, l_{10}, l_{11}, \ldots, l_9, l_{13}, l_{18}
\]
repeated 42 times (4)

Second, we use the Cut Point-Guided Search (CPGS) strategy to decide which branching node to select first. The cut points w.r.t. a pair is a sequence of control points that must be passed through when searching for a path to cover the pair. They serve as intermediate goals during the dynamic search and narrow down the search space. For example, the cut points of \(du_1(l_9, l_{17}, res)\) are \(l_4, l_8, l_9, l_{13}, l_{14}, l_{17}\). Since the path \(p\) in (4) covers the cut points \(l_4, l_8\) and \(l_9\), the uncovered cut point \(l_{13}\) is set as the next search goal. From \(p\) there are two unflipped branching nodes, 4F and 9F (denoted by their respective line numbers followed with \(T\) or \(F\) to represent the true or false branch direction). Because 9F is closer to cut point \(l_{13}\) in control flow graph than 4F, so 9F is flipped. As a result, a new test input \(t' = (x \rightarrow 0, y \rightarrow 0)\) can be generated and leads to a new path \(p' = l_{14}, l_6, l_7, l_8, l_9, l_{13}, l_{14}, l_{15}\). Now the path \(p'\) has covered the cut points \(l_4, l_8, l_9, l_{13}\) and \(l_{14}\) and the uncovered cut point \(l_{17}\) becomes the goal. From all remaining unflipped branching nodes, i.e. 4F, 13F \& 14F, the branching node 14F is chosen because it is closer than the others toward the goal. Consequently, a new test input \(t = (x \rightarrow 1, y \rightarrow 0)\) is generated which covers all cut points, and \(du_1(l_8, l_{17}, res)\) itself. Here, the cut point-guided path search takes only three iterations to cover this pair.

**CEGAR-based Data Flow Testing** The def-use pair \(du_2(l_8, l_{18}, res)\) is infeasible: if there were a test input that could reach the use, it must satisfy \(y \geq 0\) at \(l_{13}\). Since \(y\) has not been modified in the code, \(y \geq 0\) also holds at \(l_4\). As a result, \(res\) will be redefined at \(l_{10}\) since the loop guard at \(l_9\) is true. Clearly, no such a path exists for this pair which can both avoid redefinitions in the loop and reach the use. In this case, if the DSE-based approach is used, it may enter into an infinite loop-unfolding and cannot conclude the infeasibility of this pair.

To mitigate this problem, we leverage the CEGAR-based approach [18] to check feasibility. This approach starts with a coarse program abstraction and iteratively refines it. If a property violation is found, it analyzes the feasibility (i.e., is the violation genuine or the result of an incomplete abstraction?). If the violation is feasible, a counterexample path is returned. Otherwise, the proof of infeasibility is used to refine the abstraction and the checking continues. In our context, the basic idea is to encode the test requirement of a pair into the program under test and reduce the testing problem into this reachability checking problem. Figure 2 shows the transformed function \(power\) which is encoded with the test requirement of \(du_2\) in highlighted statements. We introduce a variable \(cover\_flag\) at \(l_2\). It is initialized to false and set as true immediately after the def at \(l_2\), and set to false immediately after the other definitions on variable \(res\) at \(l_{10}\). Before the use, we set a checkpoint to see whether \(cover\_flag\) is true at \(l_{14}\). If the checkpoint is unreachable, this pair can be proved infeasible. Otherwise, a counter-example, i.e. a test case that covers this pair through a def-clear path, can be generated. Here, the CEGAR-based approach can quickly conclude \(du_2\) is an infeasible pair.

**Combined DSE-CEGAR-based Data Flow Testing** From the above two examples, we can see that the DSE-based approach, as a dynamic path-based testing approach, can efficiently cover feasible pairs, while the CEGAR-based approach, as a static model checking-based approach, can eliminate infeasible ones. It is beneficial to combine the two approaches’ respective strengths to tackle the challenges in data flow testing.

The figure above shows the relation between the two approaches: The DSE-based approach is able to cover feasible pairs more efficiently (but in general, it cannot identify infeasible pairs because of path explosion) while the CEGAR-
based approach is capable of identifying infeasible pairs more effectively (it can also cover feasible pairs as well).

Figure 3 illustrates the basic workflow of the combined DSE-CEGAR approach in our data flow testing framework. The static analysis is used to find def-use pairs and their cut points from the program under test. The DSE-based approach is first used to cover as many feasible pairs as possible (within a time bound on each pair). After the DSE-based testing, for the remaining uncovered pairs, we use the CEGAR-based approach to identify infeasible pairs and cover new feasible pairs (which have not yet been covered in DSE) within a time bound. After one run of the DSE-CEGAR-based testing, we can increase the time bound for both approaches, and repeat the above process. If testing budgets permit, it can cover more feasible pairs and identify more infeasible ones. The details of these two approaches are explained in Section III.

III. APPROACH

In this section, we explain the combined DSE-CEGAR-based data flow testing framework in detail. It consists of a static analysis phase and a dynamic analysis phase.

A. Static Analysis Phase

We use standard iterative data-flow analysis algorithms [33], [34] to identify def-use pairs from the program under test (see Section IV for details).

Definition 2 (Cut Point): Given a def-use pair, its cut points are a sequence of control points that have to be passed through in succession by any control flow path covering the pair. Each control point in this sequence is called a cut point.

For illustration, consider the figure above: Let $du(l_d, l_u, x)$ be the target def-use pair, the sequence of cut points of $du$ is $l_1, l_2, l_3, l_4, l_5$ and $l_u$. Here, $l_2$ is not a cut point because a path $l_1, l_3, l_4, l_5, l_6$ can be constructed to cover the def-use pair without passing through $l_2$. The cut points of each def-use pair are computed via a context-sensitive dominator analysis [35] on the inter-procedural control flow graph.

![Algorithm 1: DSE-based Data Flow Testing](image)

**Algorithm 1: DSE-based Data Flow Testing**

**Input:** $du(l_d, l_u, x)$: a def-use pair

**Input:** $c_1, c_2, \ldots, c_n$: cut points of $du$

**Output:** input $t$ that satisfies $du$ or $nil$ if none is found

1. let $W$ be a worklist of branching nodes (initialized as empty)
2. let $t$ be an initial test input
3. repeat
   4. let $p$ be the execution path triggered by $t$
   5. if $p$ covers $du$ then return $t$
   6. $W \leftarrow W \cup \{\text{branching nodes on } p\}$
   // the redefinition pruning heuristic
   7. if variable $x$ (in $du$) is redefined after $l_d$ on $p$ then
      8. let $X$ denote the set of branching nodes after the redefinition location
      9. $W \leftarrow W \setminus X$
   10. let $t = \text{guided_search}(W)$
   11. until $t == \text{nil}$
   12. return $\text{nil}$
13.
14. **Procedure** guided_search(\textbf{reference} worklist $W$)
15. let $b'$ denote the branch to be flipped
16. if $W$ is empty then
   17. return $\text{nil}$
18. // $j$ is the index of a cut point, $d$ is the distance variable
19. \forall the branching node $b \in W$ do
   20. let $pp$ be the path prefix of $b$, i.e. $l_1, l_2, \ldots, l_b$
   // $c_1, \ldots, c_n$ are sequentially covered, while $c_i$ not yet
   21. // $i$ ← index of the uncovered cut point $c_i$ on $pp$
   // $b$ is the opposite branch of $b$
   22. if $i > j \land (i == j \land \text{distance}(b, c_i) < d)$ then
      23. $b' \leftarrow b$, $j \leftarrow i$, $d \leftarrow \text{distance}(b, c_i)$
   24. $W \leftarrow W \setminus \{b'\}$.
   // $b'$ is the opposite branch direction of the original $b'$ at $l_d^-$
25. if there exists input $t$ driving the program through $l_1, l_2, \ldots, l_{b'}^-$
   then
   26. return $t$
   27. else
   28. return guided_search($W$)

B. DSE-based Approach for Data Flow Testing

This section explains the DSE-based data flow testing approach enhanced with our cut-point-guided path search strategy. This search strategy embodies several intuitions to perform efficient path search to cover a target def-use pair. Algorithm 1 details this approach.

Algorithm 1 takes as input a target def-use pair $du(l_d, l_u, x)$ and the sequence of its cut points. If an execution path $p$ covers the target pair, the algorithm returns the input $t$ (at Line 5). Otherwise, it stores the branching nodes on $p$ into the worklist $W$, which contains all branching nodes from the explored execution paths. We first use the redefinition pruning technique (explained later) to remove invalid branching nodes (at Lines 7-9). Then we start the path search to generate a new test input in the procedure guided_search. In this procedure, we first aim
we maintain a set \( A \) of variables, \( := \) is an assignment operation. The variable definitions must occur at most one operator.

Here, \( \bullet \) is a pointer variable, \( v \) is a non-pointer variable, \( := \) is an assignment operation, and \( \wedge \) is an alias to another variable instead of \( \in \). If statement (1) is executed and \( v \) is undefined, then \( c_i \) is uncovered, then \( c_i \) is the deepest covered cut point. The intuition is that the deeper the cut point a path can reach, the closer a path toward the pair is. The cut points of a pair act as a set of search goals to follow during the dynamic search. Note the def and use of a pair also serves as cut points.

If two branching nodes have reached the same deepest cut point (indicated by \( \Rightarrow j \) at Line 22), the algorithm picks the branching node whose opposite branch has the shortest distance toward the uncovered cut point \( c_i \) (at Lines 22-23). Here we use distance\((b, c_i)\) to represent the distance between the opposite branch of \( b \) (i.e., \( b \)) and the uncovered cut point \( c_i \). The intuition is that a shorter path is easier to reach the goal. Here, the distance is approximated as the number of instructions between one statement and another. The shortest distance is the number of instructions along the shortest control flow path from the start statement to the target statement in the control flow graph \([36, 37]\). If the picked branching node can be exercised (i.e., the corresponding path constraint is satisfiable), a new test input option, where one statement contains at least one operator.

In data flow testing, due to the conservativeness of data-flow analysis, test objectives contain infeasible pairs \([13]\). In order to identify infeasible pairs, we introduce a simple but powerful encoding of data flow testing using the CEGAR-based approach. We instrument the original program \( P \) to \( P' \) and reduce the problem of test data generation to reachability checking on \( P' \). A variable cover_flag is introduced and initialized to false before the def. This flag is set to true immediately after the def and set to false immediately after the other definitions on the same variable. Before the use, we set the target predicate \( p \) as cover_flag=\( \text{true} \). As a result, if the use location is reachable when \( p \) holds, we obtain a counterexample and conclude that the pair is feasible. Otherwise, the pair is infeasible (or, since the general problem is undecidable, it does not terminate, and the result can only be concluded as unknown).

IV. IMPLEMENTATION

The data flow testing framework is built on top of a DSE engine named CAUT\([25, 28, 30]\), which includes two analysis phases: a static analysis and a dynamic analysis.

The static analysis phase collects def-use pairs, cut points, and other static program information from programs by using CIL\([\text{4}]\) (an infrastructure for C program analysis and transformation). We perform standard iterative data-flow analysis \([33, 34]\) to find intra-procedural and inter-procedural def-use pairs for C programs. We first build the control flow graph (CFG) for each function and then construct the inter-procedural control flow graph (ICFG). For each variable use, we compute which definitions on the same variable may reach this use. A def-use pair is created as a test objective for each use with its corresponding definition. We consider local and global variables in our data-flow analysis, and treat each formal parameter variable as defined at the beginning of its function, each actual parameter variable as used at its function call site \( e.g. \) library function calls), global variables as defined at the beginning of the entry function \( e.g. \) main). Following recent work on data flow testing \([39]\), we currently do not consider def-use pairs caused by pointer aliasing. Thus, we may miss some def-use pairs, but this is an independent issue and does not affect the effectiveness of our approach. More sophisticated data-flow analysis techniques \([40]\) could be used to mitigate this problem.

The dynamic analysis phase performs dynamic symbolic execution on programs. We extend the original DSE engine to whole program testing, which uses Z3\([5]\) as the constraint solver. Function stubs are used to simulate C library functions.

\[\text{Z3: http://z3.codeplex.com/}\]

\[\text{CAUT: https://github.com/tingsu/caut-lib}\]

\[\text{CIL: http://kerneis.github.io/cil/}\]
such as string, memory and file operations to improve symbolic reasoning ability. We use CIL to encode the test requirement of a def-use pair into the program under test, which is used as the input to model checkers. The DSE engine and the model checkers work on the same CIL-simplified code.

V. EVALUATION AND RESULTS

This section presents our evaluation to demonstrate the practical effectiveness of our approach for automated data flow testing. Our results on six benchmark programs show that (1) Our DSE-based approach can greatly speed up data flow testing: It reduces testing time by 60–80% than standard search strategies from state-of-the-art symbolic executors CREST [32] and KLEE [31]; and (2) Our combined approach is effective: It applies the DSE-based tool CAUT to cover as many feasible def-use pairs as possible, and then applies the CEGAR-based model checkers BLAST [20] and CPAchecker [21] to identify infeasible pairs. Overall, it reduces testing time by 40% than the CEGAR-based approach alone and improves data-flow coverage by 20% than the DSE-based approach alone.

A. Evaluation Setup

All evaluations were run on a laptop with 4 processors (2.67GHz Intel(R) i7) and 4GB of memory, running 32bit Ubuntu GNU/Linux 12.04.

Search Strategies To assess the performance of our proposed guided path search strategy for data flow testing, we choose the following search strategies to compare against:

- Random Input (RI): It generates random test inputs to drive program executions, which is a classic method to generate data-flow based test data [41].
- Random Path Search (RPS): It randomly chooses a path to exercise, which is commonly adopted in many symbolic executors [25], [31], [32].
- CFG-Directed Search (in CREST [32]): It prefers to drive the program execution down the branch with the minimal distance toward uncovered branches on the ICFG and also uses a heuristic to backtrack or restart the search under certain failing circumstances.
- RP-MD2U Search (in KLEE [31]): It uses a round-robin of a breadth-first search strategy with a Min-Distance-to-Uncovered heuristic. The breadth-first strategy favors shorter paths but treats all paths of the same length equally. The Min-Distance-to-Uncovered heuristic prefers the paths with minimal distance to uncovered statements on ICFG.
- Shortest Distance Guided Search (SDGS): It prefers to choose the path that has the shortest distance toward a target statement in order to reach the statement as quickly as possible. This strategy has been applied in single target testing [35], [37], [42]. In the context of data flow testing, the search first sets the def as the first goal and then the use as the second goal after the def is covered.

**Bechmarks** We use six benchmark programs. Four are from the Software-artifact Infrastructure Repository (SIR), including tcas, replace, printokens2 and space. They are also used in other work on data flow testing [9], [43]. We also take two industrial programs from research partners. One is an engine management system [41], [45] running on an automobile operating system (osek Os) conforming to the OSEK/VDX standard. The other is a satellite gesture control program [25] (space_control). These two industrial programs feature complicated execution logic. The detailed descriptions and statistics are listed in Table I where LOC denotes the number of lines of source code and DU the number of collected def-use pairs.

**Research Questions** In our evaluation, we intend to answer the following key research questions:

- **RQ1**: In data flow testing w.r.t. all def-use coverage, what is the performance difference among different search strategies, i.e., RI, RPS, CREST (i.e., CFG-Directed Search), KLEE (i.e., RP-MD2U Search), SDGS and CPAG (cut point-guided search) in terms of search time, program iterations and coverage level?
- **RQ2**: How effective is the combined approach (the DSE-based approach complemented with the adapted CEGAR-based approach) for data flow testing?

**Metrics and Setup** In the evaluation, we use the following metrics and experimental setup: (1) In RQ1: The search time, number of program iterations and coverage are recorded to measure the performance of different search strategies in the DSE-based approach. We target one def-use pair at a time. The coverage percentage is calculated by $C = \frac{n_{feasible}}{n_{testObj}} \times \frac{n_{infeasible}}{n_{testObj}}$. ($n_{feasible}$/$n_{infeasible}$ is the number of identified feasible/infeasible pairs, $n_{testObj}$ is the total number of pairs. In the DSE-based approach, $n_{feasible}$ is the number of covered pairs. Since this approach in general cannot identify infeasible pairs, $n_{infeasible}$ is set to 0 in coverage computation.)

The search time is calculated by $T = \sum_{i=1}^{n_{coverage}} n_{searchTime_i} / n_{searchTime_i}$ (nSearchTime_i is the time spent on the i-th covered def-use pair). The number of program iterations is calculated by $I = \sum_{i=1}^{n_{iterations}}$ (nIterations_i is the iterations of the i-th covered def-use pair). Since the testing budget (e.g., search time and program iterations) spent on an uncovered def-use pair (maybe an infeasible pair) cannot indicate which search strategy is better, only the testing budget spent on covered pairs is tabulated. The maximum search time on each pair is set to 20 seconds in case of its infeasibility. We repeat the testing process 30 times for each program/strategy and use their the average values as the final results for all measurements. The total testing time requires about two days. (2) In RQ2: Both the latest versions of the two state-of-the-art CEGAR-based model checkers, BLAST and CPAchecker are respectively used as black-box testing engines. In the evaluation, we use the following command options and configurations:

6SIR: http://sir.unl.edu/php/previewfiles.php

\begin{table}[ht]
\centering
\caption{Test Subjects From SIR and Industrial Research Partners}
\begin{tabular}{|l|l|l|l|}
\hline
Benchmark & LOC & DU & Description \\
\hline
\textbf{tcas} & 192 & 124 & collision avoidance system \\
\textbf{replace} & 384 & 285 & pattern matching and substitution \\
\textbf{printokens2} & 494 & 304 & lexical analyzer \\
\textbf{space} & 5643 & 3071 & array definition language interpreter \\
\textbf{osek_os} & 5732 & 527 & engine management system \\
\textbf{space_control} & 8763 & 1491 & satellite gesture control \\
\hline
\end{tabular}
\end{table}

7BLAST 2.7.2: http://forge.ispras.ru/projects/blast
8CPAchecker 1.3.4: http://cpachecker.sosy-lab.org/
Here, the maximum testing time on each def-use pair is set to 120 seconds. We use ocamltime, an internal script to improve memory utilization for large programs, to invoke BLAST. For CPAnceck, we use its default analysis configuration. The options --enable-recursion in BLAST and --skipRecursion in CPAnceck are both used to set recursion functions as skip. For each def-use pair, we run 30 times and use the average execution time as its final value. We run the single CEGAR-based approach and the combined DSE-CEGAR-based approach, which operates as follows: the DSE-based approach (use the cut point-guided path search strategy with the same time setting in RQ1) is first used to cover as many feasible pairs as possible; for the remaining uncovered pairs, the CEGAR-based approach is used to identify infeasible pairs and may cover some feasible pairs which have not been found by the DSE-based approach. In the evaluation, we conduct one run of the combined DSE-CEGAR-based approach.

The distribution on testing time for one pair is also given: Median, the semi-interquartile range SIQR (i.e., (Q3-Q1)/2, Q1: the lower quartile, Q2: the upper quartile) and the number of outliers which fall 3*SIQR below Q1 or above Q3.

Comparison Criterion: A better search strategy or approach in data flow testing requires less testing time, costs fewer program iterations and/or achieves higher data-flow coverage.

B. Results and Analysis

RQ1: The cut point-guided search performs the best In Table III, the performance statistics of different search strategies are listed. In Column Evaluation, RI, RPS, CREST, KLEE, SDGS, CPAnceck respectively represents the corresponding search strategies introduced before. In Column Performance Result, it shows the search time (T), program iterations (I) and data-flow coverage percentage (C). In Column Distribution, it shows the distribution on testing time for a single pair: Median (M) and SIQR. From Table III, we can see that the cut point-guided path search strategy (CPAnceck) achieves the overall best performance against the other search strategies. As expected, in general, the CPAnceck strategy requires less testing time and costs much fewer iterations than other search strategies to achieve higher data-flow coverage. It narrows the search space by following the cut points and further improves the performance by pruning redefinition paths. From Column Distribution, the median search time of one def-use pair in the CPAnceck strategy is lower than that of RP, CREST, KLEE, SDGS respectively. Thus, CPAnceck is more efficient in data flow testing.

In Figure 4, we give the column diagrams on each strategy/program in the terms of testing time, program iterations and data-flow coverage, respectively. On average, compared with CREST, KLEE and SDGS, the CPAnceck strategy reduced 69.7%, 78.6% and 39.0% in search time and 73.0%, 76.1%, 48.3% in program iterations, respectively. Compared with RPS, it roughly improves data-flow coverage by 14.7%. Compared with these novel and widely-used search strategies from many symbolic execution engines, the data clearly shows that the CPAnceck strategy is a more effective strategy for data flow testing.

However, there are some interesting phenomenons worth elaborating. RI gains higher data-flow coverage for the program printtokens2 because it is easier for RI to quickly generate random combinations of characters inputs than other path-oriented search strategies, but in general it can only cover limited number of def-use pairs. RPS is faster than all the other search strategies for the program tcas because it is a small program with finite paths and other advanced strategies incur higher path scheduling overhead. Two novel search strategies, CREST and KLEE can usually cover more pairs in most programs than RPS but they require more testing time. We note that RPS incurs much lower computation cost on path scheduling than CREST and KLEE. These two advanced strategies try to satisfy a pair by improving as much branches/statements coverage as possible, which demands more testing time. The SDGS strategy is more effective than the other three general strategies (i.e., RPS, CREST, KLEE) since it is guided by the distance metric toward the target pair. However, it is less effective than the CPAnceck strategy because the latter
and the data-flow coverage percentage of each benchmark/search strategy.

**Table III. Performance Statistics of the DSE-based, CEGAR-based, and the Combination Approach on Data Flow Testing. "DSE", "CEGAR1" and "CEGAR2" corresponding to the approach of CAUT, BLAST and CPAchecker, respectively. "-" means it does not apply.**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Approach</th>
<th>Coverage Result</th>
<th>Time Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
<td>FT</td>
</tr>
<tr>
<td>tcas</td>
<td>DSE</td>
<td>73.4%</td>
<td>2.6s</td>
</tr>
<tr>
<td></td>
<td>CEGAR1</td>
<td>72.5%</td>
<td>04m30s</td>
</tr>
<tr>
<td></td>
<td>CEGAR2</td>
<td>73.4%</td>
<td>04m30s</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR1</td>
<td>92.9%</td>
<td>04m49s</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR2</td>
<td>100.0%</td>
<td>02m58s</td>
</tr>
<tr>
<td>replace</td>
<td>DSE</td>
<td>60.0%</td>
<td>05m28s</td>
</tr>
<tr>
<td></td>
<td>CEGAR1</td>
<td>65.9%</td>
<td>01h23m</td>
</tr>
<tr>
<td></td>
<td>CEGAR2</td>
<td>61.0%</td>
<td>01h09m</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR1</td>
<td>78.5%</td>
<td>31m02s</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR2</td>
<td>77.5%</td>
<td>29m58s</td>
</tr>
<tr>
<td>printtokens2</td>
<td>DSE</td>
<td>36.9%</td>
<td>55.8s</td>
</tr>
<tr>
<td></td>
<td>CEGAR1</td>
<td>38.7%</td>
<td>01h45m</td>
</tr>
<tr>
<td></td>
<td>CEGAR2</td>
<td>55.4%</td>
<td>02h55m</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR1</td>
<td>70.9%</td>
<td>19m12s</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR2</td>
<td>77.1%</td>
<td>31m51s</td>
</tr>
<tr>
<td>space</td>
<td>DSE</td>
<td>30.1%</td>
<td>10m37s</td>
</tr>
<tr>
<td></td>
<td>CEGAR1</td>
<td>32.7%</td>
<td>24h37m</td>
</tr>
<tr>
<td></td>
<td>CEGAR2</td>
<td>35.7%</td>
<td>26h50m</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR1</td>
<td>37.5%</td>
<td>01h32m</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR2</td>
<td>38.1%</td>
<td>03h47m</td>
</tr>
<tr>
<td>osek_os</td>
<td>DSE</td>
<td>71.6%</td>
<td>04m47s</td>
</tr>
<tr>
<td></td>
<td>CEGAR1</td>
<td>87.9%</td>
<td>04m19s</td>
</tr>
<tr>
<td></td>
<td>CEGAR2</td>
<td>95.2%</td>
<td>05m35s</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR1</td>
<td>93.9%</td>
<td>10m02s</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR2</td>
<td>97.1%</td>
<td>14m52s</td>
</tr>
<tr>
<td>space_control</td>
<td>DSE</td>
<td>64.3%</td>
<td>04m47s</td>
</tr>
<tr>
<td></td>
<td>CEGAR1</td>
<td>94.8%</td>
<td>05h23m</td>
</tr>
<tr>
<td></td>
<td>CEGAR2</td>
<td>94.3%</td>
<td>07h31m</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR1</td>
<td>94.8%</td>
<td>01h26m</td>
</tr>
<tr>
<td></td>
<td>DSE+CEGAR2</td>
<td>94.3%</td>
<td>44m59s</td>
</tr>
</tbody>
</table>

contains more optimization techniques.

**RQ2: The combined approach is effective** Table III lists the performance statistics of different approaches in data flow testing. In Column Evaluation, DSE, CEGAR1 and CEGAR2 represents the single testing approach from CAUT, BLAST and CPAchecker, respectively. DSE+CEGAR1/CEGAR2 represents...
the combined approach of DSE and CEGAR. Column Coverage Result lists the data-flow coverage ($C$) and the coverage details ($CD$) (the number of feasible/infeasible/unknown pairs. If a testing approach cannot give a definite conclusion on the feasibility of a pair within the given time bound, we call this pair unknown). Column Time Result lists the testing time spent on feasible pairs ($FT$), infeasible pairs ($IT$) and total pairs ($TT$). Note the total testing time $TT$ is the sum of $FT$, $IT$ and the time spent on unknown pairs, so $TT$ should be longer than $TT + FT$. In Column Distribution, $M_f$ and $M_t$ represent the median testing time (in seconds) on identifying a feasible and an infeasible pair, respectively, $S$ stands for SIQ and $O$ is the number of outliers in the CEGAR-based approach.

Note, in general, the DSE-based approach can only identify the feasible and unknown (i.e., uncovered) pairs (so we treat the uncovered pairs as unknown pairs) while the CEGAR-based approach can identify both feasible and infeasible pairs.

From Table III we can observe that the DSE-based approach can cover a large portion of feasible pairs detected by the CEGAR-based approach (see Column $CD$). Moreover, by comparing the testing time spent on feasible pairs between the DSE-based and the CEGAR-based approach (see Column $FT$), we can see that the DSE-based approach is very effective in covering feasible pairs. A reasonable explanation is that the DSE-based approach is a dynamic explicit path-based testing method while the CEGAR-based approach is a static model checking-based testing method. The static approach requires much higher testing overhead. On the other hand, it is easier for the CEGAR-based approach to identify infeasible pairs (see Column $IT$) while the DSE-based approach has to check all possible paths before confirming which pairs are infeasible. So it is beneficial to combine the DSE-based approach with the CEGAR-based approach to gain their respective advantages i.e., reduce testing time on feasible pairs and improve coverage by eliminating infeasible pairs.

In Figure 5 we present the column diagrams of the total testing time (normalized in percentage), the data-flow coverage and the coverage details of the DSE-based, CEGAR-based, and the combined approach. In detail, the combination strategy can on average reduce total testing time by 40% than the CEGAR-based approach alone. It can also eliminate infeasible pairs more easily and on average improve data-flow coverage by 20% than the DSE-based approach alone. Thus, the combined approach can provide a more practical way of data flow testing.

In Table III we also find that the testing performances of the two model checkers and their conclusions on the number of feasible/infeasible pairs have some differences within the constrained testing time (see Column $CD$). A reasonable explanation is that their underlying constraint solvers, implementation languages, search heuristics have different impact on testing performance. They also have different performances on programs exhibiting different features. For example, in tcas, all program inputs are integral, while space control involves much floating-point computation and many struct/union manipulations. In addition, the number of infeasible pairs detected by BLAST is generally fewer than that of CPAchecker (see Column $CD$). BLAST is slightly faster in identifying feasible pairs while CPAchecker can usually identify infeasible pairs more quickly (see Column $M_f$ and $M_t$).

Discussion We have developed a simple, yet powerful method to reduce data flow testing into model checking and used two CEGAR-based state-of-the-art model checkers to evaluate its practicality. From the evaluations on two combination instances (CAUT+BLAST and CAUT+CPAchecker), we observe a consistent trend: the combined DSE-CEGAR-based approach can greatly reduce testing time and improve data-flow coverage than the two approaches alone. In addition, we have also used the DSE and CEGAR-based approach to cross-check each other by comparing their results on the same def-use pairs. This helps to validate the correctness and consistency of both techniques and also make our testing results more reliable.

C. Threats to Validity

First, we implemented all search strategies on our CIL-based tool CAUT. The original RP-MD2U strategy in KLEE uses the number of LLVM instructions to measure the minimal distance between one instruction to another while CAUT uses CIL instructions as its distance metric. KLEE is a static symbolic executor, while CAUT is a dynamic symbolic executor. These differences may affect the performance of the RP-MD2U strategy on the benchmarks. In addition, the implementation of the two search strategies from CREST and KLEE may differ from their original versions. For these threats, we carefully inspected the relevant source code and technical reports [46] to ensure our implementation conformance and correctness. The decision to engineer the data flow testing framework on our DSE-based tool is based on the following considerations: (1) CREST does not support real numbers, composite structures or pointer reasoning, but these features are required in testing real-world programs; and (2) KLEE is a static symbolic executor, thus it is more difficult to reason about possible variable redefinitions caused by variable aliasing than dynamic symbolic executors. Implementing all search strategies on top of the same tool provides the convenience to record and compare the testing performances of different search strategies.

Second, in our current implementation, we do not identify def-use pairs caused by pointer aliasing in the risk of missing some def-use pairs. More sophisticated static or dynamic analysis techniques [40] could be adopted to identify more def-use pairs. However, we believe that this is an independent issue and not the focus of the work. The effectiveness of our DSE-based and CEGAR-based approach should remain.

As for possible external threats, we have evaluated our approach on a small set of benchmarks, which include programs used in previous data flow testing research as well as industrial programs. From these programs, the effectiveness of our approach is evident. Although it is interesting to consider additional test subjects, due to our novel, general methodologies, we believe that the results should be consistent.

VI. RELATED WORK

This section discusses three strands of related work: (1) data flow testing, (2) DSE-based advanced coverage testing, and (3) directed symbolic execution.

Data Flow Testing Data flow testing has been empirically demonstrated to be more effective [6], [9] than other structural testing techniques, but with much higher testing cost [12], [13]. Much research has considered how to aid data flow testing. Some efforts use random testing combined with program slicing [41], while a few others use the collateral coverage relationship [47], [48] between branch/statement and data-flow criteria to generate data-flow test data. There is also work that applies search-based techniques [39], [49]–[51] to perform data
flow testing. For example, Ghiduk et al. [49] use a genetic algorithm, which takes as input an initial population of random inputs, and adopts a designated fitness function [52] to measure the closeness of execution paths against a target def-use pair. It then uses genetic operations (e.g., selection, crossover and mutation) to search the next promising input to cover this pair. Other efforts include Nayak et al. [50], who use a particle swarm optimization algorithm, and Ghiduk [51], who uses the ant colony algorithm to derive test data for the target def-use pair. Recently, Vivanti et al. [59] also use a genetic algorithm to conduct data flow testing on classes [7]. They use a fitness function to guide the search to reach the def and then the use. In contrast, we adopt an enhanced dynamic symbolic execution technique to perform data flow testing, and demonstrate how to combine our DSE-approach with our CEGAR-based approach to effectively deal with infeasible test objectives, which has not been investigated in prior work.

Classic symbolic execution [38] is also used to statically select control flow paths to do data flow testing. Buy et al. [53] combine data-flow analysis, symbolic execution and automated deduction to perform data flow testing on classes. Symbolic execution first identifies the relation between the input and output values of each method in a class, and then collects the method preconditions from a feasible and def-clear path that can cover the target pair. An automated backward deduction technique is later used to find a sequence of method invocations (i.e., a test case) to satisfy these preconditions. However, little evidence is provided on the practicality of this approach. Hong et al. [54] use a model checking approach to generate data-flow oriented test data. It models the program as a Kripke structure and characterizes data-flow criteria via a set of CTL property formulas. A counterexample for a property formula represents a test case for a specific def-use pair. However, this method requires manual annotation with unclear scalability since it is evaluated on only a single function. Baluda et al. [55] use a combined method of concolic execution [32] and abstract refinement [56] to compute accurate branch coverage. It refines the abstract program from a sequence of failed test generation operations to detect infeasible branches. In contrast, we directly encode test objectives into the program under test and use interpolation-based model checkers (different from their refinement method) as black-box engines, which is fully automatic and flexible.

**Advanced Coverage Testing via DSE.** Extensive work exists to apply the DSE-based technique [14], [15], [28], [31] for test case generation w.r.t. certain advanced coverage criteria. Bardin et al. [57] propose a label coverage criterion to imply a number of advanced criteria (MC/DC and weak mutation criteria). This label coverage criterion is both expressive and amenable to efficient automation. However, it cannot handle those criteria that impose constraints on paths (e.g., data-flow criteria) rather than program locations. Pandita et al. [23] propose a trade-off approach to achieve a specified coverage criterion through source code transformations. The block coverage in the transformed program implies the MC/DC coverage in the original program. Augmented DSE [26] enforces advanced criteria such as boundary, logical and mutation criteria by augmenting path conditions with additional conditions. However, in this paper, we aim to automate data flow testing, which has not been considered before in the context of DSE. We have designed an efficient search strategy to find paths that cover def-use pairs.

**Directed Symbolic Execution** Much research [36], [37], [42], [58], [59] has been done to guide path search toward a specified program location in symbolic execution. Do et al. [59] make use of data dependency analysis [60] to guide the search process to a program location of interest, while we use a dominator analysis. Ma et al. [37] propose a call chain backward search heuristic to find a feasible path, backward from the target program location to the entry. However, it is difficult to adapt this approach on data flow testing, because it requires that a function can be decomposed into logical parts when the target locations (e.g. the def and the use) are located in the same function [61]. But decomposing a function itself is a nontrivial task. Zamfir et al. [60] narrow the path search space by following a limited set of critical edges and a statically-necessary combination of intermediate goals. On the other hand, our approach finds a set of cut points from the program entry to the target locations, which makes path exploration more efficient. Xie et al. [58] integrate fitness-guided path search strategy with other heuristics to reach a program point. The proposed strategy is only efficient for those problems amenable to its fitness functions. Marinescu et al. [42] use a shortest distance-based guided search method (like the adapted SDGS heuristic in our evaluation) with other heuristics to quickly reach the line of interest in patch testing. In comparison, we combine several search heuristics to guide the path exploration to traverse two specified program locations (i.e. the def and use) sequentially for data flow testing.

**VII. CONCLUSION**

We have proposed a combined symbolic execution and model checking approach to automate data flow testing. First, we have adapted dynamic symbolic execution (DSE) for data flow testing and introduced a novel path search strategy to make the basic approach practical. Second, we have devised a simple encoding of data flow testing via counterexample-guided abstraction refinement (CEGAR). The two approaches offer complementary strengths: DSE is more effective at covering feasible def-use pairs, while CEGAR is more effective at rejecting infeasible pairs. Indeed, evaluation results have demonstrated that their combination reduces testing time by 40% than the CEGAR-based approach alone and improves coverage by 20% than the DSE-based approach alone. This work not only provides novel techniques for data flow testing, but also suggests a new perspective on this problem to benefit from advances in symbolic execution and model checking. In further work, we would like to apply this data flow testing technique on larger programs and make deeper combinations between DSE and CEGAR, e.g., learning some predicates from DSE to help CEGAR avoid unnecessary explorations and save testing time.

**ACKNOWLEDGMENT**

We would like to thank the anonymous reviewers for their valuable feedback. Ting Su is partly supported by ECNU Project of Funding Overseas Short-term Studies, Domestic Academic Visit and International Conference and NSFC Project No. 91118007. Jifeng He is partially supported by NSFC Project No. 61361136002 and Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (ZF1213). Zhoulai Fu and Zhendong Su are partially supported by United States NSF Grants 1117603, 1319187, and 1349528.


