SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

March 2023, Vol. 66 132101:1–132101:21
https://doi.org/10.1007/s11432-020-3403-2

BATON: symphony of random testing and concolic testing through machine learning and taint analysis

Bihuan CHEN[1,2*], Yang LIU[3], Xin PENG[1,2], Yijian WU[1,2] & Shengchao QIN[4]

[1]*School of Computer Science, Fudan University, Shanghai* 201203*, China;*
[2]*Shanghai Key Laboratory of Data Science, Fudan University, Shanghai* 201203*, China;*
[3]*School of Computer Science and Engineering, Nanyang Technological University, Singapore* 639798*, Singapore;*
[4]*School of Computing, Teesside University, Middlesbrough* TS1 3BX*, UK*

Received 22 July 2020/Revised 15 October 2021/Accepted 9 December 2021/Published online 11 November 2022

Abstract Random testing is scalable but often fails to hit corner program behaviors, while systematic testing (e.g., concolic execution) is promising to cover corner program behaviors but is not scalable to explore all program behaviors. Prior attempts to integrate random testing with systematic testing lack targeted guidance. In this paper, we propose a guided hybrid testing approach, named BATON, to synergize random testing with concolic testing. It integrates the knowledge inside test cases and their executions into a conditional execution graph, and uses such knowledge to guide test case generation. Specifically, we learn classification models for some conditionals in the conditional execution graph in a demand-driven way. These models are used to guide random testing to reach and cover partially-covered conditionals. We further employ targeted concolic testing to cover conditionals that cannot be fully covered by guided random testing. We implemented BATON for Java and evaluated it on three benchmarks. The results show that BATON improved branch coverage and mutation score over random testing by 16.2%–29.4% and 19.0%–30.0%, over adaptive random testing by 16.8%–33.8% and 19.4%–34.2%, over concolic testing by 2.3%–29.9% and 2.9%–30.1%, and over simple hybrid testing by 1.6%–14.5% and 1.4%–18.7%.

Keywords system testing, random testing, concolic testing

Citation Chen B H, Liu Y, Peng X, et al. BATON: symphony of random testing and concolic testing through machine learning and taint analysis. Sci China Inf Sci, 2023, 66(3): 132101, https://doi.org/10.1007/s11432-020-3403-2

1 Introduction

Automated software testing is one of the primary ways to detect bugs. A scalable method is random testing [1, 2]. However, randomly-generated test cases may lack diversity and only cover a small part of all possible behaviors of a program, which hinders the effectiveness of random testing. Then, adaptive random testing [3, 4] and Quasi-random testing [5, 6] are proposed to improve the diversity of generated test cases in order to cover different program behaviors. The main challenges of these techniques are that they lack insightful guidance during test case generation and hence generate redundant test cases; and they can efficiently cover general conditionals (e.g., $x > 100$), but often fail to cover corner conditionals (e.g., $x == 1000$), and thus cannot break through the coverage plateau [7].

Different from random testing, systematic testing leverages the knowledge inside a program to generate test cases. Symbolic execution [8–10] and concolic execution [11, 12] are typical systematic testing techniques. They collect path conditions based on the internal code structure, and systematically negate and solve path conditions to generate test cases that execute different paths. In that sense, they are naturally promising to cover corner conditionals. However, given the exponential program paths and limited time budget, they usually end up with a small number of explored paths. Their main challenge is that they require extensive constraint solving, rely on the capability of constraint solvers and might get stuck in loops, and thus become inefficient or even impractical for complex programs.

* Corresponding author (email: bhchen@fudan.edu.cn)

To integrate the efficiency of random testing on general conditionals with the effectiveness of systematic testing on corner conditionals, some prior attempts have been made. At the system testing level, Majumdar and Sen [13] proposed to use concolic execution to search for an uncovered branch when random testing saturates; and Driller [14] applied the similar idea to binary fuzzing. At the unit testing level, Inkumsah and Xie [15] and Galeotti et al. [16] integrated search-based testing with symbolic execution; and Garg et al. [17] interleaved guided random testing with concolic execution. These approaches leverage concolic/symbolic execution in a blind or less targeted way, and thus expensive concolic/symbolic execution is excessively used, even for general conditionals. Their key challenge is how to leverage dynamic testing execution knowledge from one technique to guide test generation in another technique.

To address the challenges, we focus on system testing in this paper, and propose a guided hybrid testing approach, named Baton, to improve code coverage and bug detection. Baton synergizes random testing with concolic testing and systematically guides test case generation based on the knowledge inside test cases and their executions. While we are not the first one to use random testing and concolic testing in combination [13], the targeted guidance as proposed in Baton is novel and improves testing effectiveness.

In particular, to obtain the knowledge inside test case executions, for each executed test case, we record the conditional execution sequence (i.e., a sequence of executed conditionals) via instrumentation, identify the dependency between executed conditionals and input variables via dynamic taint analysis, and integrate them into a conditional execution graph. This graph captures various knowledge: the set of test cases executing the true/false branch of a conditional, the dependency of a conditional on input variables, the coverage of a conditional (i.e., whether it is partially or fully covered), and the context of a conditional (i.e., the prefix conditionals to reach it).

Choosing a partially-covered conditional in the conditional execution graph as the target, we leverage machine learning in a demand-driven way to learn classification models for the target and its prefix conditionals based on two kinds of knowledge: the set of test cases executing the true/false branch of the conditional, and the dependency between the conditional and input variables. Therefore, we learn a two-class classification model for a fully-covered conditional that can label a test case as taking either the true branch or false branch, and a one-class classification model for a partially-covered conditional that can indicate whether or not a test case takes the covered branch. Since such models capture the constraints of conditionals, we use them to guide random testing to reach and fully cover the target.

If the target is not fully covered by our guided random testing, we employ targeted concolic testing to fully cover it based on the set of test cases that execute the true or false branch of the target. Specifically, we run concolic execution with one of the test cases, and negate the path condition at the target to generate test cases by constraint solving. Such test cases can fully cover the target. The collected symbolic constraints for each conditional along the path are used as features to improve classification models for guided random testing.

We implemented Baton for Java and conducted experiments on three benchmarks from the (adaptive) random testing community [18,19], the concolic/symbolic execution community [20,21], and the software testing community [22]. We compared Baton with random testing, adaptive random testing, concolic testing, and simple hybrid testing. On average, Baton can improve branch coverage and mutation score over random testing by 16.2%–29.4% and 19.0%–30.0%, over adaptive random testing [4] by 16.8%–33.8% and 19.4%–34.2%, over concolic testing [20] by 2.3%–29.9% and 2.9%–30.1%, and over simple hybrid testing [13] by 1.6%–14.5% and 1.4%–18.7%.

In summary, this paper makes the following contributions.

• We propose a guided hybrid testing approach, named Baton, to synergize random testing with concolic testing with targeted guidance.

• We propose a conditional execution graph to integrate the knowledge inside test cases and their executions for guiding Baton.

• We have implemented Baton for Java and evaluated Baton on several benchmarks with respect to branch coverage and mutation score, which has shown promising results.

The rest of the paper is organized as follows. Section 2 introduces a motivating example and the approach overview. Section 3 elaborates the approach details. Section 4 evaluates the approach. Section 5 reviews the related work, before Section 6 draws the conclusion.

```java
public class Foo {
  public void bar(double x, double y, int m, int n) {
    x = MultiTainter.taintedDouble(x, 1);
    y = MultiTainter.taintedDouble(y, 2);
    m = MultiTainter.taintedInt(m, 4);
    n = MultiTainter.taintedInt(n, 8);
    if (x == 0.0 || y == 0.0) {  // C0
      Profile.add(0, true, TYPE.if);
      if (m >= 0) {  // C1
        Profile.add(1, true, TYPE.if);
        // do something
      } else {
        Profile.add(1, false, TYPE.if);
        // do something
      }
      if (n >= 0) {  // C2
        Profile.add(2, true, TYPE.if);
        // do something
      } else {
        Profile.add(2, false, TYPE.if);
        // do something
      }
    } else {
      Profile.add(0, false, TYPE.if);
      double t = Math.sqrt(Math.abs(1 / y - 1 / x));
      if (t < 0.01) {  // C3
        Profile.add(3, true, TYPE.if);
        exe(t, n);
      } else {
        Profile.add(3, false, TYPE.if);
        // do something
      }
    }
  }
  private void exe(double t, int n) {
    for (int i = 1; i <= 10000; i++) {  // C4
      Profile.add(4, true, TYPE.for);
      t += t * i / (Math.abs(n) + 1) ;
      if (t > 1.0) {  // C5
        Profile.add(5, true, TYPE.if);
        // do something
      } else {
        Profile.add(5, false, TYPE.if);
        // do something
      }
    }
    Profile.add(4, false, TYPE.for);
  }
  public static void main(String[] args) {
    new Foo().bar(1, 1, 1, 1);
  }
}
```

Figure 1 (Color online) A running example (with instrumented code in blue).

2 Overview

In this section, we first introduce a motivating example and then present the overview of Baton.

2.1 Motivating example

Figure 1 presents a Java program, which will be used as a running example to illustrate the main ideas of Baton throughout the paper. The program has six conditionals `C0`, `C1`, ..., `C5`. Random testing [1] and adaptive random testing [4] can easily cover `C3`, `C4`, and `C5`, but fail to cover the true branch of `C0` and cannot reach `C1` and `C2` as `C0` is a corner conditional that is very difficult to satisfy. Thus, (adaptive) random testing has a branch coverage of 57.1%. However, concolic testing [20] can easily cover `C0`, `C1`, and `C2`, but may fail to cover `C3` due to the complex non-linear constraint of `C3`, leading to a branch coverage of 64.3%. Even though some meta-heuristic-based solvers (e.g., CORAL [23]) have been developed for complex constraints, they are non-deterministic and might be time-consuming. As a result, concolic testing may get stuck in the loop, i.e., keep unfolding the loop that has a complex non-linear conditional `C5`. Hence, simple hybrid testing [13] may be stuck in the loop without reaching the uncovered `C0`, leading to a branch coverage of 57.1%. Instead, by synergizing random testing with concolic testing in a targeted and guided way, Baton starts with random testing to cover `C3`, `C4`, and `C5`, switches to concolic testing to efficiently break through `C0`, and switches back to random testing and guides random testing to cover

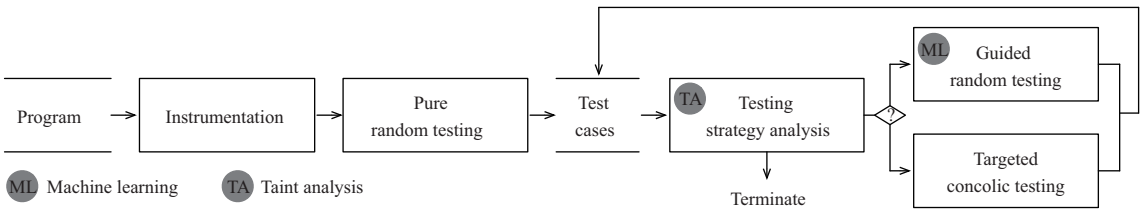

Figure 2 The approach overview of Baton.

C1 and C2. Thus, Baton has a much higher branch coverage of 100%.

2.2 Approach overview

Figure 2 shows an overview of Baton, which takes as input the program under test, and returns a set of automatically generated test cases. The overall algorithm of Baton is described in Algorithm 1.

Baton first performs instrumentation on the program under test at source code level to facilitate the collection of the dynamic execution knowledge of test cases (line 3, see Subsection 3.1). Specifically, we add a branch recorder for each branch of the conditionals in the program, which is used to record the conditional execution sequence (including the taken branch of conditionals) for each test case; and set a taint tag for each input variable of the program, which is used to identify the dependency of conditionals on input variables.

Then, Baton applies pure random testing to generate the first set of n test cases (line 4) because there is no prior knowledge for guiding either random testing or concolic testing. After that, Baton works through a number of iterations (lines 5–15).

At each iteration, Baton performs testing strategy analysis based on the collected execution knowledge; i.e., it chooses a testing strategy (either guided random testing or targeted concolic testing) to cover a target partially-covered conditional (lines 7–15, see Subsection 3.2). Specifically, we collect two kinds of execution knowledge: the conditional execution sequence by running each generated test case, and the taint tags for each executed conditional (i.e., the dependency of each conditional on input variables) through dynamic taint analysis (line 7, see Subsection 3.2.1). Based on such execution knowledge and the test cases, we construct a conditional execution graph, which maintains various knowledge for each conditional (line 8, see Subsection 3.2.2). Then, we systematically choose a partially-covered conditional from the conditional execution graph as the target for testing, where we give at most m chances for each partially-covered conditional to be the target (line 9, see Subsection 3.2.3). Finally, we determine a testing strategy to fully cover the target (lines 12 and 14, see Subsection 3.2.4).

If the target conditional has been chosen for less than m times, Baton employs our guided random testing to attempt to fully cover the target (lines 12 and 13, see Subsection 3.3). Here we try guided random testing for several (at most $m-1$) times to cover a particular target conditional because of its efficiency. In particular, we extract a trace from the start conditional of the conditional execution graph to the target conditional. Then, based on the set of test cases executing the true and false branches of a conditional, and the dependency of a conditional on input variables, we learn a two-class classification model for each fully-covered conditional in the trace, which labels a generated test case as taking the true or false branch; and we learn a one-class classification model for the target conditional, which indicates whether a generated test case takes the covered branch or not. We learn the models in such a demand-driven way that only the conditionals in the trace are involved in the learning. Based on such models, we generate test cases that can follow the trace, reach the target conditional and cover its uncovered branch.

Otherwise, if the target conditional has been chosen for m times, Baton employs our targeted concolic testing to attempt the fully cover the target (lines 14 and 15, see Subsection 3.4). Hence, due to its effectiveness, targeted concolic testing is only used for once after our guided random testing has failed to fully cover the target for $m-1$ times. In particular, we run concolic execution with a test case that reaches the target, negate the path condition at the target conditional, and apply constraint solving to generate test cases that cover the uncovered branch of the target, if feasible. We collect the symbolic constraint for each executed conditional along the path, and attach it to the conditional execution graph. It is used as a feature to improve learned classification models.

Baton terminates when (i) the time budget is reached, or (ii) all the partially-covered conditionals in the conditional execution graph have been attempted to cover by Baton for m times (line 10).

Algorithm 1 The overall algorithm of Baton

Input: P: the program under test.
Output: T: the set of generated test cases.
1: $T \leftarrow \emptyset$; `// initialize the set of generated test cases`
2: $G \leftarrow$ null; `// initialize the conditional execution graph`
3: IP $\leftarrow$ instrument the program under test P; `// instrumentation`
4: NT $\leftarrow$ generate n test cases using pure random testing; `// pure random testing`
5: **while** true **do**
6: $T \leftarrow T \cup$ NT; `// testing strategy analysis`
7: $S, D \leftarrow$ collect execution knowledge for each test case in NT;
8: $G \leftarrow$ construct the conditional execution graph using NT, S, and D;
9: $c \leftarrow$ choose a partially-covered conditional, which has been chosen as the target for no more than m times, from G as the target;
10: **if** time budget is reached, or c is null **then**
11: **break** `// terminate Baton`
12: **else if** this is not the m-th time that c is chosen as the target **then**
13: NT $\leftarrow$ generate at most n test cases using our guided random testing; `// guided random testing`
14: **else if** this is the m-th time that c is chosen as the target **then**
15: NT $\leftarrow$ generate test cases using our targeted concolic testing; `// targeted concolic testing`
16: **return** T.

3 Methodology

In this section, we first elaborate the details of Baton, and then discuss the interplay between different techniques in Baton.

3.1 Instrumentation

We instrument the program under test at source code level to collect dynamic execution knowledge of each test case. We consider two kinds of dynamic knowledge: the conditional execution sequence, and the dependency between conditionals and input variables. For the ease of presentation, hereafter we focus our discussion on `if` and `for` conditionals. `switch` conditional and `?:` operator are handled in the similar way to `if` conditional. `foreach`, `while` and `do` conditionals are handled in the similar way to `for` conditional.

On one hand, to collect the conditional execution sequence, we traverse the abstract syntax tree of the program under test, assign an id and determine the type (e.g., `if`, `for` and `while`) for each encountered conditional, and insert a branch recorder for each branch of the conditional. A branch recorder is an API call `Profile.add (id,branch,type)` that maintains the sequence of executed conditionals including the taken branch of each executed conditional.

Example 1. The program in Figure 1 has six conditionals. For the `if` conditional at line 7, its id is `0`, and its two branch recorders are inserted at lines 8 and 24 to capture whether the true or false branch is taken. The conditional at line 36 is a `for` conditional, its id is `4`, and its two branch recorders are inserted at lines 37 and 47, capturing whether the loop body is executed or the loop is exited.

On the other hand, to track the dependency between conditionals and input variables, we set a taint tag for each input variable of the program under test. As we focus on system testing, we identify the set of input variables by identifying all the primitive arguments of all the method invocations in the entry method `main`. Besides, we provide users with a configuration capability to specify and customize which arguments of a method need to set a taint tag. For each input variable, we insert an API call to set a taint tag at the beginning of the corresponding method, which follows the exposed interface of the dynamic taint analysis tool Phosphor [24] we used in Baton.

Example 2. The program in Figure 1 has four primitive arguments in the method invocation in the `main` method, i.e., the four arguments of the `bar` method. Thus, it has four input variables `x`, `y`, `m`, and `n`, whose tags are respectively set to `1`, `2`, `4`, and `8` at lines 3–6.

3.2 Testing strategy analysis

For each generated test case, we analyze its execution knowledge for testing strategy analysis in three steps (Subsections 3.2.2–3.2.4).

3.2.1 *Execution knowledge*

We first define execution knowledge. For simplicity, a test case, t, is denoted as an assignment to input variables. The conditional execution sequence of a test case t, S_t, is defined as a list of pairs $\langle \text{id}, \text{branch} \rangle$, where id is the id for each executed conditional, and branch $\in \{T, F\}$ means the true or false branch is taken. S_t is collected by running t with instrumented program.

For each executed conditional c in S_t, the dependency on input variables, d_t^c, is defined as the subset of input variables the executed conditional has data flow dependency on. d_t^c is context-sensitive, and the context is the prefix of c in S_t. Hence, we can distinguish the dependency information of a conditional that is executed for multiple times (e.g., a method is invoked in multiple places of the program under test). Therefore, the dependency information from a test case t, D_t, is denoted as a list of dependencies of executed conditionals in S_t on input variables. D_t is collected via dynamic taint analysis on the test case t with the instrumented program.

Example 3. For a randomly generated test case $t_1 = \langle 3257.8, 7136.2, 6, -1525 \rangle$, its conditional execution sequence S_{t_1} is $\langle 0, F \rangle, \langle 3, F \rangle$; and the dependency information D_{t_1} is $d_{t_1}^0 = \{x, y\}, d_{t_1}^3 = \{x, y\}$. D_{t_1} shows that conditional `C0` at line 7 and `C3` at line 26 all depend on `x` and `y`.

3.2.2 *Constructing conditional execution graph*

To integrate execution knowledge of different test cases, we incrementally construct a conditional execution graph according to each generated test case t, the conditional execution sequence S_t and dependency information D_t. In a conditional execution graph G, each node v corresponds to an executed conditional, and is denoted by a 5-tuple $\langle \text{id}, \text{type}, \text{dep}, e_T, e_F \rangle$, where id and type is the id and type of the executed conditional, dep is the dependency of the executed conditional on input variables, and e_T and e_F are edges in the graph. Each edge e_T or e_F represents the execution of a conditional i by taking the true or false branch to reach another conditional j, and is denoted by a 3-tuple $\langle p, q, \Sigma \rangle$, where p and q are nodes ($p.\text{id} = i$ and $q.\text{id} = j$) in the graph, and Σ is a set of test cases that take the edge. The start node of G represents the conditional that is always first executed in all conditional execution sequences.

Algorithm 2 gives the procedure to update a conditional execution graph G. It first gets the root node (initially a newly-created node without setting its id, type, dep, e_T and e_F) of G as the current node in the traversal (line 1), and then works by iterating a conditional execution sequence S_t of a test case t (lines 4–30). At each iteration, it has three steps. First, it gets the id of the executed conditional and its taken branch (line 5), sets the id and type of the current node if this is the first time to execute the conditional under a context[1)] (i.e., the prefix trace of the conditional in the graph) (lines 6 and 7), and updates the dependency information (line 8). Second, it decides if the current node corresponds to a loop conditional (e.g., `for` and `while`). If yes, it pushes the current node to a stack if this is the first time and first iteration to execute the loop (lines 9 and 10), and it pops a node from the stack if the loop is exited (i.e., the false branch is taken) (lines 11 and 12). We maintain such a stack for loop conditionals that loops are not unfolded in the graph (see the following step). Third, it links the current node to the next node (line 19/28) if the current node's true/false branch is not taken before (lines 13 and 14/22 and 23). Here the next node is set to the peek node of the stack if the next conditional in S_t corresponds to the peek node (i.e., this is not the first iteration of the loop) (lines 15 and 16); otherwise, it is set to a newly-created node (line 18). It then attaches the test case t to the taken branch of the current node (line 20/29), and sets the current node to the next node following the taken branch to continue the traversal (line 21/30).

Apart from test cases and dependency information, a conditional execution graph also maintains coverage (i.e., whether a conditional is partially-covered or fully-covered) and context (i.e., the prefix trace of a conditional in the graph to reach the conditional) knowledge for each executed conditional. Compared to a symbolic execution tree whose size can be exponential to loop iteration bounds as loops are unfolded till a bound is reached, we do not unfold loops in a conditional execution graph but only distinguish conditional executions inside loops. Thus, the context knowledge is not precise, and a conditional execution graph is designed to balance scalability (making the size irrelevant to loop iterations) and context precision (see the evaluation in Subsection 4.6).

Example 4. Given t_1, S_{t_1}, and D_{t_1} from Example 3, the conditional execution graph after executing t_1 is shown in Figure 3(a), where the dep of each node is shown in braces and the Σ of each edge is omitted

1) Throughout the paper, conditional means context-sensitive conditional when it is referred in a conditional execution graph.

Algorithm 2 Update a conditional execution graph

Input: G: the graph; t: test case; S_t: conditional execution sequence; D_t: dependency information.
Output: G: the updated graph.

```
1: curr ← the root node of G; // the current node
2: next ← null; // the next code
3: stack ← ∅; // a stack of loop-related nodes
4: for i ← 0 to S_t.size() do
5:    id ← S_t[i].id; branch ← S_t[i].branch;
6:    if curr.id = null then
7:       curr.id ← id; curr.type ← the type of conditional id;
8:    curr.dep ← curr.dep ∪ D_t[i];
9:    if curr.type is loop-related and curr ≠ stack.peek() then
10:      stack.push(curr);
11:   if curr.type is loop-related and branch = F then
12:      stack.pop();
13:   if branch = T then
14:      if curr.e_T = null then
15:         if S_t[i + 1].id = stack.peek().id then
16:            next ← stack.peek();
17:         else
18:            next ← newNode(); // uninitialized
19:         curr.e_T ← newEdge(curr, next);
20:      curr.e_T.Σ ← curr.e_T.Σ ∪ {t};
21:      curr ← curr.e_T.q;
22:   else if branch = F then
23:      if curr.e_F = null then
24:         if S_t[i + 1].id = stack.peek().id then
25:            next ← stack.peek.();
26:         else
27:            next ← newNode(); // uninitialized
28:         curr.e_F ← newEdge(curr, next);
29:      curr.e_F.Σ ← curr.e_F.Σ ∪ {t};
30:      curr ← curr.e_F.q;
31: return G.
```

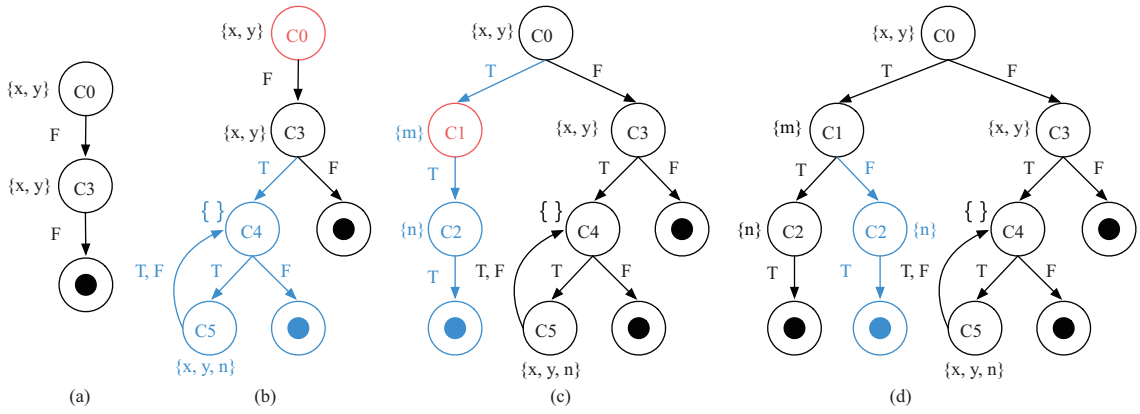

Figure 3 (Color online) Conditional execution graph of Figure 1. (a) After executing t1; (b) after executing t2; (c) after concolic testing; (d) after mutation.

for clarity. For test case $t_2 = \langle 6218.4, 6204.4, -8416, -4772\rangle$, it takes the true branch of `C3` and executes the loop at lines 36–47. Figure 3(b) is the updated conditional execution graph for t_2, where the new part is highlighted in blue. The loop conditional `C4` does not depend on input variables as the number of loop iterations is a constant. The conditional `C5` inside the loop depends on `x`, `y`, and `n`, and both its true and false branch are taken during loop iterations in t_2. Since loops are not unfolded, there are two edges from the node for `C5` back to the node for `C4`, and we do not create new nodes for `C4` and `C5` for each iteration of the loop.

Note that loops with `return` statements, `break` statements and recursive invocations will break Algorithm 2. To handle them, we add instrumentations to mark `return` and `break` statements as well as the entries and exits of recursions, which are not introduced in Subsection 3.1 and not reflected in Algorithm 2 for simplicity.

3.2.3 *Choosing target conditional*

In a conditional execution graph, we choose a target conditional for BATON to cover. The target conditional needs to be partially-covered and depend on input variables. If a conditional is partially-covered but does not depend on input variables, we cannot fully cover it no matter how we generate test cases. An `if` conditional is said to be fully-covered if both its true and false branches have been taken; otherwise, it is partially-covered. A `for` conditional is said to be fully-covered if its loop body has been executed zero times (i.e., directly taking the false branch without executing the loop body) and one or more times (i.e., taking the true branch to execute the loop body); otherwise, it is partially-covered.

To systematically choose a target conditional, we perform breadth-first search on the conditional execution graph, starting from the start node. In this way, we give higher priority to shallow conditionals in the graph than deep conditionals. The reasons are that deep conditionals are more difficult to be fully covered than shallow conditionals, and covering shallow conditionals can reveal deep conditionals. Moreover, instead of infinitely attempting to cover a target conditional, we give at most m attempts; i.e., we will not choose a conditional as the target if it has been chosen for m times.

Example 5. In Figure 3(b), the node for conditional `C0` is the start node. Conditional `C3` and `C5` is fully-covered; `C0` is partially-covered as its true branch is not taken; and `C4` is partially-covered as the loop is not directly exited without executing its body. Assuming that it is the first time to choose a target conditional, the start node is chosen as the target, highlighted in red in Figure 3(b). `C4` will never be chosen as the target conditional as it does not depend on input variables.

3.2.4 *Choosing testing strategy*

After a target conditional c is chosen, we choose our guided random testing (Subsection 3.3) to cover its uncovered branch if c has been chosen as the target for less than m times; otherwise (which means guided random testing has failed to fully cover c for $m-1$ times), we choose our targeted concolic testing (Subsection 3.4) to cover the uncovered branch. Hence, we give $m-1$ attempts to our guided random testing to fully cover the target before we finally resort to our targeted concolic testing for once, considering the efficiency of guided random testing and the effectiveness of target concolic testing. If targeted concolic testing also fails to cover the target, we regard the target as infeasible (although the target can be feasible but concolic testing cannot solve the path condition), and will not attempt to cover it anymore.

3.3 Guided random testing

Given the target conditional in the conditional execution graph, we attempt to guide random testing to reach and fully cover it based on the knowledge in the conditional execution graph. Our overall idea is to learn a classification model for a conditional to determine how a generated test case will take its branches, and to use a search-based way to generate test cases that have a high potential to cover a specific branch. The learned models are used in the search to measure how close a generated test case covers a branch.

First, we extract a trace by backward traversal from the target conditional to the start node of the conditional execution graph. We maintain the depth of each node to the start node, traverse the conditional execution graph in such a backward way that the depth always decreases, and thus generate only one trace. Hence, for loop conditionals, we do not consider how many iterations the loop is executed before reaching the target for the same reason of not unfolding loops in the conditional execution graph (Subsection 3.2.2). The trace gives the information on how to reach the target conditional, while the target conditional gives the information on where to cover. The trace is denoted as $v_0 \xrightarrow{e_{b_0}} v_1 \xrightarrow{e_{b_1}} \cdots v_n \xrightarrow{e_{b_n}} v_t$, where v_0 is the start node, v_t is the target conditional, and $b_i \in \{T, F\}$ $(0 \leqslant i \leqslant n)$ means that v_i takes the edge e_{b_i} (i.e., the true or false branch).

Second, for each fully-covered conditional v in the trace, if v has dependency on input variables (v.dep! = null), we will learn a two-class classification model, which can label a generated test case as taking the true or false branch. If v does not depend on input variables, we will not learn such a model because we cannot control its taken branch by manipulating the values of input variables. On the other hand, for the partially-covered target conditional v_t, we will learn a one-class classification model, which can indicate whether a generated test case takes the covered branch or not. For the other partially-covered conditionals in the trace, we will not learn such a model due to the systematic way we choose

the target conditional (Subsection 3.2.3); i.e., they have already been chosen as the target, and their uncovered branches have been regarded as infeasible.

• To build the dataset for learning a two-class classification model of a conditional v, we use the test cases that execute the true and false branch of v ($v.e_T.\Sigma$ and $v.e_F.\Sigma$). If v is an `if` conditional, $v.e_T.\Sigma$ (resp. $v.e_F.\Sigma$) is used to create and label instances as taking the true (resp. false) branch. If v is a `for` conditional, $v.e_T.\Sigma$ (resp. $v.e_F.\Sigma - v.e_T.\Sigma$) is used to create and label instances as executing the loop body (resp. directly exiting the loop).

• To create an instance based on a test case, we consider two kinds of features: (1) the value of the input variables that v depends on (v.dep), and (2) the satisfaction (0 or 1) of the collected symbolic constraints of v (Subsection 3.4). The second feature exists only for conditionals executed by concolic execution. By these features, we attempt to learn the constraint of v in a clasification model.

Notice that the one-class classification model for the target conditional is similarly learned. We learn the models in such a demand-driven way that only the conditionals in the trace are involved in the learning. The machine learning models we use are evaluated in Subsection 4.2.

Third, we generate test cases that can follow the trace and cover the uncovered branch of v_t in a search-based way.

• Seed. We use a combination of randomly generated test cases and the previously executed test cases that reach v_t as the seed. The size of the seed is limited to n, i.e., the same size of test cases generated by pure random testing (line 4 in Algorithm 1).

• Objectives. The searching objectives are: (1) a test case takes e_{b_i} of v_i ($0 \leqslant i \leqslant n$ and a model has been learned for v_i) as many as possible and as close as possible, and (2) a test case takes the uncovered branch of v_t as close as possible. Note that the learned models output a value in [0,1] for each test case, measuring how close each test case covers a branch.

• Mutation. We mutate the value of one of the input variables (i.e., $v_0.\text{dep} \cup v_1.\text{dep} \cup \cdots \cup v_n.\text{dep} \cup v_t.\text{dep}$) that the trace depends on, which can reduce the searching space.

In each iteration of the multi-objective searching process, we mutate each test case in the seed, create an instance for the mutated test case, and classify the instance using those learned models to evaluate the objectives. The test cases in the seed and the mutated test cases that have higher value of the objectives are kept as the seed for the next iteration. After a number of iterations, the set of Pareto-optimal test cases (whose size can be at most n) are generated.

Example 6. Following Example 5, the start node in Figure 3(b) is the target conditional. Therefore, we only learn a one-class classification model for `C0`, and generate test cases that are classified as not taking the false branch of `C0`. However, as `C0` is a corner conditional that is difficult to satisfy, our guided random testing fails to cover its true branch, and hence target concolic testing is used to generate two test cases $\langle 0.0, 0.0, 0, 0\rangle$ and $\langle -4232.6, 0.0, 8529, 5555\rangle$ (as will be discussed in Example 7). The conditional execution graph is then updated to Figure 3(c), where `C1` is chosen as the target conditional. Then, we learn a two-class classification model for `C0` and a one-class classification model for `C1`. Mutating one of `x`, `y`, and `m` that `C0` and `C1` depend on, we generate test cases that take the false branch of `C2`; e.g., one of the test cases $\langle 0.0, 0.0, -3194, 0\rangle$ is generated from $\langle 0.0, 0.0, 0, 0\rangle$ by mutating `m`, and the updated graph is in Figure 3(d), where there are two nodes for `C2` as we distinguish possible paths to a conditional for providing targeted guidance.

3.4 Targeted concolic testing

Given the target conditional that our guided random testing fails to cover, we run concolic execution with one of the test cases that reaches the target conditional, backtrack the symbolic execution tree, negate the path condition whenever encountering the target conditional, and apply constraint solving to generate a test case, if feasible, to cover the uncovered branch. More than one test case might be generated, as a compound conditional corresponds to several bytecode-level conditionals and a conditional inside a loop might be encountered for several times during the backtrack. In this targeted way, we only need to run one concolic execution to generate one symbolic path, without an exhaustive depth-first path exploration to first find an uncovered branch that is unknown in advance as is done in [13]. In addition, we collect the symbolic constraints for each executed conditional along the path, which are more accuracy representations of a conditional, and are used as features (Subsection 3.3) to improve the learned model (see the evaluation in Subsection 4.2). Besides, the generated test cases can help guided random testing to explore deeper (see Example 6).

Example 7. Following Example 5, the start node in Figure 3(b) is the target conditional, but it is not covered by guided random testing as shown in Example 6. Thus, targeted concolic testing is run with the test case $t_1 = \langle 3257.8, 7136.2, 6754, -1525\rangle$ in Example 3; and two test cases $\langle 0.0, 0.0, 0, 0\rangle$ and $\langle -4232.6, 0.0, 8529, 5555\rangle$ are generated and the symbolic constraints of `C0` and `C3` are collected. The constraints of `C0` are $x == 0$ and $y == 0$, which correspond to the two bytecode-level conditionals as `C0` is a compound conditional.

3.5 The interplay

Baton integrates several techniques, i.e., random testing, concolic testing, machine learning, and taint analysis. The interplay between these techniques, which distinguishes Baton from the existing studies (see Section 5), is discussed as follows. In general, as illustrated in Figure 2, machine learning and taint analysis are supporting techniques to provide targeted guidance to random testing and concolic testing.

More specifically, taint analysis guides random testing and concolic testing to focus on conditionals that have dependency on input variables, and avoids wasting testing effort on input-independent conditionals (Example 5). Taint analysis also provides machine learning with representative features (i.e., the input variables that a conditional depends on). Machine learning, together with the conditional execution graph, guides random testing to systematically put testing effort on different program behaviors (Example 6).

Random testing, with the guidance of machine learning and taint analysis, efficiently reaches commonly-executed program behaviors that might be difficult for concolic testing to touch, and thus relieves some of the burden from concolic testing (Example 6). Moreover, random testing provides useful test cases that reach a target conditional to achieve a targeted concolic testing (Example 7). On the other hand, concolic testing breaks through the coverage bottleneck, and provides more representative features (i.e., the collected symbolic constraints of a conditional) for machine learning to improve the accuracy of the learned models and thus improve the guidance to random testing. Further, concolic testing provides useful test cases to help guided random testing explore deeper (Example 6).

4 Evaluation

We implemented Baton for Java with 7.4k lines of Java code. We implemented our targeted concolic testing by extending jDart [20], used Weka [25] to learn classification models, and used Phosphor [24] for dynamic taint analysis.

4.1 Evaluation setup

To evaluate the effectiveness of Baton, we compared Baton with several state-of-the-art testing tools with respect to branch coverage and fault detection rate on several benchmarks. We used EclEmma [26] to compute branch coverage. We used mutation score as the indicator of fault detection rate. As evidenced in [27], mutants can be used as a substitute for real faults when comparing testing techniques, and mutation score can be an indicator of fault detection rate. We used the mutation testing system muJava [28] to automatically generate mutants by applying all method-level mutation operators to all classes of the used benchmarks and then to compute the mutation score. Assertions for mutation testing are manually written for each benchmark program. We ran our experiments on a desktop with 3.50 GHz Intel Xeon CPU and 16 GB RAM.

Baton is most closely related to random testing, adaptive random testing, and concolic testing, and thus we compared Baton with them. Previous empirical studies [4, 19, 29] have shown that adaptive random testing methods FSCS [30], RRT [31], and EAR [4] have similar effectiveness. Hence, we selected EAR as the state-of-the-art adaptive random testing method. For concolic testing, we used jDart [20] as the-state-of-the-art as it outperformed other symbolic/concoloc execution engines (e.g., SPF [10] and jFuzz [32]). When using jDart, we used Z3 [33] and Coral [23] as the constraint solver. We also compared Baton with the simple or unguided hybrid testing approach in [13], which is the closest work to ours. Since it targets C programs and the tool is not available, we implemented a Java version based on our testing infrastructure of Baton. This simple hybrid testing can be seen as Baton without the guidance of machine learning and taint analysis. Notice that unit testing tools (e.g., Randoop [34] and EvoSuite [35]) generate tests against each method separately, but Baton generates system tests that test the entire system as a whole. As a result, some branches that will never be covered in system tests could be covered

Table 1 Benchmark programs

Program	Lines of code	Source
Bessj	131	(adaptive) random testing community [18, 19]
Expint	86	(adaptive) random testing community [18, 19]
Fisher	71	(adaptive) random testing community [18, 19]
Gammq	89	(adaptive) random testing community [18, 19]
Remainder	48	(adaptive) random testing community [18, 19]
Triangle	26	(adaptive) random testing community [18, 19]
Triangle2	46	(adaptive) random testing community [18, 19]
WBS	231	Concolic/symbolic execution community [20, 21]
Raytrace	570	Concolic/symbolic execution community [20, 21]
MinePump	559	Concolic/symbolic execution community [20, 21]
Siena	1256	Concolic/symbolic execution community [20, 21]
NanoXML	4608	Concolic/symbolic execution community [20, 21]
Schedule	412	Siemens benchmark [22]
Schedule2	374	Siemens benchmark [22]
Totinfo	315	Siemens benchmark [22]
PrintTokens2	570	Siemens benchmark [22]
Replace	564	Siemens benchmark [22]

by unit tests. Thus, the comparison between Baton and Randoop/EvoSuite becomes biased, and we did not conduct such comparison, following [4, 13, 20]. Potential extensions to Baton for supporting unit testing will be discussed in Subsection 5.2.

The evaluations were conducted on the following benchmarks. A benchmark from the (adaptive) random testing community [18, 19] contains 11 numerical programs, and we selected 7 programs that involve complex mathematical computations; and the remaining four programs were excluded as they were not challenging for all testing techniques. A benchmark from the concolic/symbolic execution community [20, 21] has 5 systems: WBS is a wheel brake system; Raytrace is a system for rendering shades on surfaces; MinePump is a real-time system that monitors and controls the fluid level and methane concentration in a mine shaft; Siena is an Internet-scale event notification middleware; and NanoXML is an XML parser for Java. NanoXML takes XML input, while Baton currently supports primitive types for input variables (Subsection 3.1). Following the same procedure in [21], we treat an XML as an array of chars. The Siemens benchmark [22], widely used in the testing community, has 7 programs, and we selected 5 of them and omitted TCAS and PrintTokens as they are easy to cover. The Siemens programs were originally written in C and were manually translated to Java in [21]. These benchmarks were chosen to ensure the diversity (from different communities) of programs under test. The detailed information of these benchmark programs is reported in Table 1, including the number of lines of code and the benchmark source.

We ran the state-of-the-art random testing (RT), adaptive random testing (ART) [4], concolic testing (CT) [20], and simple hybrid testing (HT) [13] with the same amount of time that Baton took; and Baton was configured to use the second stopping criterion and empirically set the number of generated tests n and the number of attempts m to 10 and 3 respectively (Subsection 2.2). To account for the randomness, we ran each method for each benchmark for five times and reported their average results in the following subsections. Based on the above setup, we conducted the evaluation to answer four research questions:

- **Q1:** How do different classification models impact on Baton?
- **Q2:** Can Baton improve branch coverage and mutation score over RT, ART, CT, and HT?
- **Q3:** How do guided random testing, targeted concolic testing and taint analysis contribute to branch coverage and mutation score?
- **Q4:** What is performance overhead of Baton?

4.2 Impact of classification models (Q1)

We used naive Bayes, random forest, LibSVM, and J48 as the classifiers to learn the classification models in our guided random testing; and we performed 10-fold cross validation and computed F-measure [36] as the indicator of the accuracy of learned models. As shown in Figure 4(a), naive Bayes, random forest, and

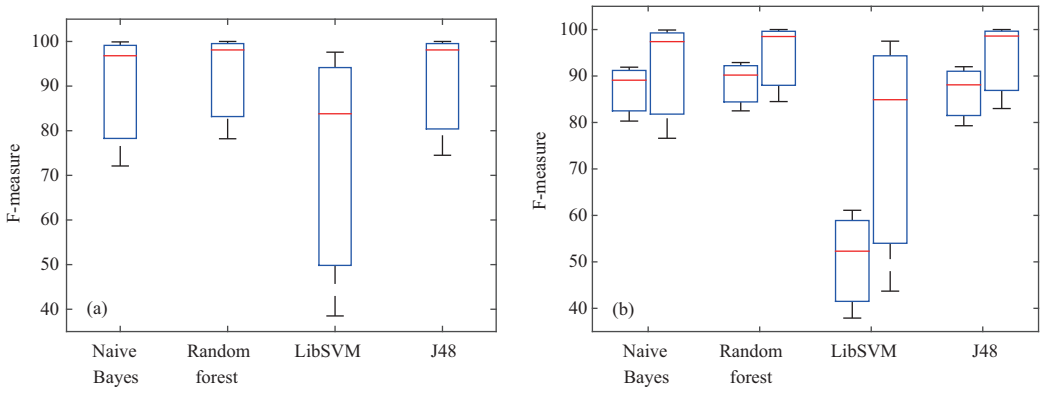

Figure 4 (Color online) Accuracy of learned models. (a) Different classifiers; (b) with/without symbolic constraints.

J48 achieved similar accuracy, while LibSVM was 13% lower on average. Notice that consistent results were obtained for precision and recall, and thus here, we only reported F-measure. Meanwhile, Baton achieved almost the same branch coverage and mutation score with these four classifiers, but Baton with LibSVM invoked our targeted concolic testing 4% more often than Baton with the other classifiers. The two differences of LibSVM from the other classifiers are statistically significant ($p \leqslant 0.05$ across all benchmark programs using Wilcoxon signed-rank test [37]). These results indicate that the low accuracy of learned models causes unnecessary invocations of targeted concolic testing to cover conditionals that could have been covered by guided random testing. In terms of performance, with random forest, LibSVM and, J48, our guided random testing took 1.6×, 2.2×, and 0.2× more time than with naive Bayes. Overall, naive Bayes and J48 are better classifiers for Baton, and thus we show the results with native Bayes in the following subsections.

Moreover, we also analyzed the impact of using collected symbolic constraints as features on the accuracy of learned models. From the previous 10-fold cross validation, we distinguish learned models that have and do not have symbolic constraints as features, and report their F-measure in the right and left boxplot in Figure 4(b). We can see that for each classifier, the accuracy is at least 8% higher when symbolic constraints are used as features. This result indicates that the collected symbolic constraints are not the only key features that ensure the effectiveness of our machine learning step.

Further, we also analyzed the size of instances used to learn classification models. Generally, the size of instances has a positive correlation with the number of generated tests (see the penultimate column of Table 2) and a negative correlation with the depth of a conditional located in the conditional execution graph. Numerically, the mean size of instances for the three benchmarks is respectively 59, 1006, and 4076. This further indicates that naive Bayes is suitable for Baton, as it suits well for small dataset [38].

Answer to Q1. Naive Bayes is the most suitable classifiers for Baton by considering the accuracy of learned models, invocations of targeted concolic testing, performance, and size of instances; and symbolic constraints can improve the accuracy of learned models.

4.3 Branch coverage and mutation score (Q2)

Table 2 reports the results of branch coverage and mutation score with RT, ART, CT, HT, and Baton. The first and second columns list the benchmarks and programs. The third to fifth columns report the program details: the lines of code, the number of non-loop/loop conditionals, and the number of generated mutants. The next fifteen columns report the branch coverage, the mutation score, and the number of generated test cases of the five testing methods. The last column gives the time overhead of Baton.

RT and ART had the lowest branch coverage and mutation score for all programs except for Siena since both RT and ART do not have the capability to break through corner conditionals. ART, claimed to be better than RT, was unexpectedly worse than RT, because of the heavy calculation of distances among test cases in ART, which has also been evidenced in [18] and reflected by the much smaller number of generated test cases. Baton significantly improved branch coverage and mutation score over RT (p-value is 0.00044 and 0.0003 using Wilcoxon signed-rank test [37]) by 29%, 16%, 28% and 30%, 19%, 23% for those three benchmarks, and over ART (p-value is 0.00044 and 0.0003) by 32%, 17%, 34% and 34%, 19%, 29%. This is because Baton integrates concolic testing to break through corner conditionals and

Table 2 Comparisons of Baton with RT, ART, CT, and HT on branch coverage and mutation score[a)]

	Program				Branch coverage (%)					Mutation score (%)					Generated tests (#)					Time (s)
	Name	LOC	Cond.	Mut.	RT	ART	CT	HT	Baton	RT	ART	CT	HT	Baton	RT	ART	CT	HT	Baton	
B.M. 1 [18,19]	Bessj	131	9/2	1572	47.2	44.4	38.9	84.2	**91.7**	42.3	31.7	37.7	49.6	**82.0**	115740	2220	7	171	562	58.8
	Expint	86	7/3	709	33.3	30.6	47.2	66.7	**75.0**	22.0	20.9	48.7	52.9	**63.5**	44610	1200	7	121	122	17.2
	Fisher	71	6/2	866	81.2	68.8	56.2	62.5	**100**	43.8	28.2	21.1	35.5	**76.8**	49970	830	4	148	366	13.8
	Gammq	89	9/3	849	52.4	52.4	26.2	52.4	**57.1**	73.5	71.4	22.4	57.5	**75.3**	25800	920	4	48	78	10.8
	Remainder	48	5/4	577	79.2	79.2	37.5	81.9	**91.7**	72.1	72.1	29.8	69.4	**72.3**	9230	400	31	65	67	2.4
	Triangle	26	6/0	242	68.8	68.8	**100**	75.0	**100**	33.9	33.9	**84.3**	64.5	**84.3**	24100	570	19	132	108	3.1 (1.0)*
	Triangle2	41	10/0	458	47.1	47.1	**100**	91.2	**100**	27.1	27.1	70.1	65.0	**70.7**	8200	330	14	130	60	2.2 (0.7)*
	Average				58.5	55.9	58.0	73.4	**87.9**	45.0	40.8	44.9	56.3	**75.0**	39664	924	12	116	195	–
B.M. 2 [20,21]	WBS	225	37/0	810	37.8	37.8	**66.7**	**66.7**	**66.7**	24.2	24.2	45.7	**48.8**	**48.8**	85810	1250	24	604	2638	29.3 (0.7)*
	Raytrace	363	21/2	1601	57.7	57.7	65.4	71.2	**75.0**	36.6	36.6	50.3	54.9	**55.5**	58070	760	90	235	424	70.5
	MinePump	366	32/1	455	42.3	42.3	**61.5**	**61.5**	**61.5**	21.1	21.1	**53.4**	**53.4**	**53.4**	65230	1230	36	561	5555	90.9 (0.9)*
	Siena	1256	163/14	1758	**16.1**	**16.1**	13.2	**16.1**	**16.1**	52.6	52.6	49.1	52.4	**52.7**	202190	3990	130480	1290	7850	561.5
	NanoXML	4608	199/37	3088	5.1	2.3	19.0	16.5	**20.9**	7.0	4.8	23.4	20.1	**26.2**	74600	2070	31710	1158	1590	126.1
	Average				31.8	31.2	45.2	46.4	**48.0**	28.3	27.9	44.4	45.9	**47.3**	97180	1860	32468	770	3611	–
B.M. 3 [22]	Schedule	303	19/6	645	37.1	37.1	80.6	72.6	**87.1**	31.3	23.7	59.1	50.5	**67.3**	23820	1120	121	452	1488	21.8 (1.3)*
	Schedule2	334	29/6	762	58.1	58.1	**81.4**	77.9	**81.4**	43.3	43.3	73.6	60.2	**74.3**	134220	2790	134	663	4434	115.8 (14.5)*
	Totinfo	189	16/10	1490	77.8	75.9	79.6	75.9	**87.0**	82.3	82.0	81.8	81.8	**83.7**	33240	860	5	114	333	14.0
	PrintTokens2	529	96/8	1663	72.2	51.9	**86.6**	80.6	**86.6**	62.4	51.1	63.9	56.8	**71.3**	427990	9830	489	1232	28745	879.4 (4.3)*
	Replace	665	127/11	3289	22.1	16.7	**68.8**	63.0	66.3	23.0	15.7	**64.0**	60.9	62.2	46470	1540	1715	2500	2352	46.0 (6.3)*
	Average				53.5	47.9	79.4	74.0	**81.7**	48.5	43.2	68.5	62.0	**71.8**	133149	3228	493	992	7470	–

a) The highest values are highlighted in bold.

explores more program behaviors. Besides, Baton generated 203×, 27×, and 18× less test cases than RT because the targeted guidance in Baton helps systematically put testing effort on different conditionals. Baton generated a comparable number of test cases to ART. Notice that both RT and ART became saturated in branch coverage within the same time that Baton took.

Compared to CT, for programs that have floating-point inputs and complex computations (e.g., Bessj, Expint, Fisher, Gammq, Remainder, Raytrace, and Totinfo), Baton achieved much higher branch coverage and mutation score. This is because the constraint solver used in CT takes much time or is unable to handle some complex floating-point constraints and CT may get stuck. However, Baton uses random testing to cover complex conditionals as many as possible, and only applies concolic testing for those corner complex conditionals. For programs that have many input-dependent loops (e.g., Siena and NanoXML), Baton had slightly higher branch coverage and mutation score than CT. The reason is that CT can get stuck in such loops (i.e., keep unfolding the loops), which is also reflected in a large number of generated test cases. Instead, Baton does not unfold loops but only distinguishes conditional executions inside loops, and thus will not get stuck in loops without making any further progress. For programs where the inputs are integer and most computations are simple (e.g., Triangle, Triangle2, WBS, MinePump, Schedule, Schedule2, PrintTokens2, and Replace), Baton achieved the almost same branch coverage and mutation score as CT. In such cases, CT took much less time than Baton, as indicated by a $*$ mark in the last column where the time overhead of CT is given in parentheses. This is because the constraint solver can efficiently solve almost all the constraints. Baton becomes less helpful for such programs. This gives us the implication that a static analysis to estimate the complexity of the program under test or even the complexity of the conditionals can be helpful to decide which testing method should be used to cover which program or conditional in a proactive way. Overall, the improvement of Baton over CT on branch coverage and mutation score is significant, i.e., p-value is 0.00578 and 0.0012.

HT achieved higher branch coverage and mutation score than RT, ART, and CT, but lower than Baton for those complex programs (e.g., Bessj, Expint, Fisher, Gammq, Remainder, and Raytrace). The reason is that, like Baton, HT applies concolic testing to only break through corner conditionals; but different from Baton, it does not distinguish the same conditional under different contexts and needs to search for uncovered conditionals during concolic testing. For less complex programs (e.g., Triangle, Triangle2, Schedule, Schedule2, PrintTokens2, and Replace), HT had lower branch coverage and mutation score than both CT and Baton. This is because CT considers almost all the contexts of each conditional, but HT does not distinguish the context of conditionals. Overall, the improvement of Baton over HT on branch coverage and mutation score is significant (p-value is 0.00096 and 0.00064). This indicates that our guided random testing and targeted concolic testing via learning and taint analysis are effective.

Answer to Q2. Baton can significantly outperform RT, ART, CT, and HT in terms of branch coverage and mutation score (p-values are less than the significant level 0.05). It owes to the use of a conditional execution graph to guide random testing in a learning and context-sensitive way and the use of targeted concolic testing.

Table 3 Unique conditionals (fully/partially) covered and unique mutants killed

Program	Unique conditionals (#/#)										Unique mutants (#)									
	Baton vs. RT		Baton vs. ART		Baton vs. CT		Baton vs. HT		Baton vs. ALL		Baton vs. RT		Baton vs. ART		Baton vs. CT		Baton vs. HT		Baton vs.ALL	
	RT	Baton	ART	Baton	CT	Baton	HT	Baton	ALL	Baton	RT	Baton	ART	Baton	CT	Baton	HT	Baton	ALL	Baton
Bessj	0/0	7/2	0/0	8/2	0/0	8/2	0/0	2/0	0/0	1/0	25	649	0	790	28	724	9	433	53	411
Expint	0/0	7/2	0/0	8/2	1/0	7/2	0/0	3/1	1/0	3/1	12	306	9	311	161	266	13	88	161	87
Fisher	0/0	3/0	0/0	5/0	0/0	4/0	0/0	3/0	0/0	2/0	21	307	1	422	7	489	23	370	38	265
Gammq	0/0	2/1	0/0	2/1	0/0	5/6	0/0	2/1	0/0	2/1	16	31	11	44	1	450	10	60	17	31
Remainder	0/0	3/0	0/0	3/0	0/0	9/1	0/0	4/0	0/0	2/0	9	10	9	10	0	245	0	53	9	2
Triangle	0/0	2/0	0/0	2/0	0/0	0/0	0/0	1/0	0/0	0/0	0	122	0	122	1	1	0	48	1	1
Triangle2	0/0	8/1	0/0	8/1	0/0	0/0	0/0	2/0	0/0	0/0	4	204	4	204	6	9	15	40	15	0
WBS	0/0	5/9	0/0	5/9	0/0	0/0	0/0	0/0	0/0	0/0	0	207	0	207	0	10	0	0	0	0
Raytrace	0/0	5/0	0/0	5/0	0/0	4/0	0/0	1/0	0/0	1/0	0	190	0	190	4	72	4	11	5	6
MinePump	0/0	4/4	0/0	4/4	0/0	0/0	0/0	0/0	0/0	0/0	0	59	0	59	0	0	0	0	0	0
Siena	0/0	0/0	0/0	0/0	0/0	14/0	0/0	0/0	0/0	0/0	0	1	1	2	5	66	4	9	8	0
NanoXML	0/0	36/13	0/0	42/14	0/0	4/1	0/0	10/3	0/0	2/0	0	420	0	437	24	48	5	159	28	24
Schedule	0/0	8/12	0/0	8/12	0/0	0/2	0/0	2/3	0/0	0/2	13	241	0	277	1	54	3	110	17	49
Schedule2	0/0	8/5	0/0	8/5	0/0	0/0	0/0	1/1	0/0	0/0	0	236	0	236	12	17	2	109	13	9
Totinfo	0/0	5/0	0/0	6/0	0/0	4/0	0/0	6/0	0/0	4/0	11	31	10	35	1	29	8	36	11	28
PrintTokens2	0/0	15/4	0/0	29/10	0/0	0/0	0/0	4/2	0/0	0/0	66	185	35	327	2	124	2	219	92	20
Replace	0/0	31/53	0/0	36/58	3/0	0/0	3/0	2/0	3/0	0/0	2	1293	0	1531	62	3	106	150	111	0

4.4 Unique conditionals and mutants (Q2)

We also looked into the conditionals that were uniquely covered by one testing method but not by the others as well as the mutants that were uniquely killed. Table 3 reports the detailed results. The first column lists the programs. The next ten columns report the number of unique conditionals that are fully/partially-covered when comparing Baton with RT, ART, CT, HT, and all of them (ALL). In the same way, the next ten columns give the number of unique mutants that are killed.

Compared to RT and ART, Baton covered all the conditionals that were covered by RT and ART, and covered many conditionals that were only partially covered or even not reached by RT and ART. This is because RT and ART have difficulty for corner conditionals, which are covered by Baton through concolic testing. Besides, only a small number of mutants were killed by RT and ART but not killed by Baton, while the number of mutants that were killed by Baton but not killed by RT and ART was very large. This indicates that Baton integrates and guides random testing in such a reasonable way that it takes almost full advantage of random testing (see more discussions in Subsection 4.5).

Compared to CT, Baton covered all the conditionals that were covered by CT for all programs except for Expint and Replace. Specifically, the nondeterministic constraint solver CORAL [23] caused the one uncovered conditional in Expint, and our imprecise handling of loop conditionals (i.e., not unfolding loops) caused those three uncovered conditional in Replace. Besides, Baton covered many unique conditionals, especially for those complex programs. With respect to unique mutants, except for Expint and Replace, CT killed a small number of unique mutants, but Baton killed many unique mutants. This indicates that our integration of concolic testing is also reasonable (see more discussions in Subsection 4.5).

The comparison results between HT and Baton are similar to those between CT and Baton, which indicates that simple integration of random testing and concolic testing is not sufficient to take the full advantage of random testing and concolic testing. Interestingly, when compared to all the covered conditionals and killed mutants of RT, ART, CT, and HT, Baton covered more conditionals and killed more mutants, especially for complex programs. This indicates that our integration of random testing and concolic testing is not just a simple combination but a synergy to further improve the effectiveness.

Answer to Q2. Baton can improve branch coverage and mutation score over RT, ART, CT, and HT thanks to the reasonable synergy of random testing and concolic testing via targeted guidance.

4.5 Contributions of each component (Q3)

To investigate how guided random testing, targeted concolic testing, and taint analysis contribute to branch coverage and mutation score, we ran Baton in two configurations: with taint analysis enabled and disabled, respectively. In each configuration, we computed the number of test cases generated by guided random testing and targeted concolic testing as well as the branch coverage and mutation score achieved by these test cases. This experiment was run on the first benchmark as the interplay between the components can be better reflected by complex benchmarks. Table 4 reports the results, where columns BC and MS show branch coverage and mutation score.

With taint analysis disabled, both guided random testing and targeted concolic testing spent more time for all programs except for Remainder and Triangle. The reason is that those input-independent

Table 4 Statistics about each component in Baton with taint analysis enabled or disabled

Subject	Taint analysis enabled								
	Guided random testing				Targeted concolic testing				Strategy analysis
	Time (s)	Tests (#)	BC (%)	MS (%)	Time (s)	Tests (#)	BC (%)	MS (%)	Time (s)
Bessj	2.8	555	47.2	34.4	47.6	7	88.9	81.8	8.4
Expint	0.9	100	55.6	46.7	15.8	22	69.4	61.4	0.5
Fisher	2.7	358	81.2	30.9	9.3	8	100	74.6	1.8
Gammq	0.8	74	52.4	42.3	8.2	4	38.1	63.1	1.8
Remainder	0.7	63	79.2	70.5	1.5	4	45.8	30.7	0.2
Triangle	1.2	97	68.8	28.5	1.5	11	100	84.3	0.4
Triangle2	0.9	52	61.8	39.3	1.1	8	79.4	59.2	0.2
Average	1.4	186	63.7	41.8	12.1	9	74.5	65.0	1.9
Subject	Taint analysis disabled								
	Guided random testing				Targeted concolic testing				Strategy analysis
	Time (s)	Tests (#)	BC (%)	MS (%)	Time (s)	Tests (#)	BC (%)	MS (%)	Time (s)
Bessj	3.2	797	47.2	34.4	54.7	10	86.1	74.7	1.3
Expint	1.8	130	55.6	46.7	18.8	23	69.4	61.4	0.2
Fisher	2.9	405	81.2	30.9	11.6	7	93.8	60.9	0.4
Gammq	1.2	99	47.6	40.5	14.5	4	38.1	63.1	0.1
Remainder	0.6	65	79.2	70.5	1.5	4	45.8	30.7	0.1
Triangle	1.2	101	68.8	28.5	1.5	11	100	84.3	0.1
Triangle2	1.7	100	61.8	39.3	1.8	8	79.4	59.2	0.1
Average	1.8	242	63.1	41.5	15.0	10	73.2	62.0	0.3

conditionals will be chosen as the target conditional for guided random testing and targeted concolic testing to cover as we assume that all conditionals depend on all input variables when taint analysis is disabled. Such testing effort is actually wasted, and also leads to a larger number of test cases for guided random testing. Besides, strategy analysis took less time because taint analysis was disabled, but the reduction was overshadowed by the increased overhead of guided random testing and targeted concolic testing. Furthermore, the branch coverage and mutation score of guided random testing on Gammq were decreased when taint analysis was disabled due to the imprecise features for learning, while the branch coverage and mutation score of targeted concolic testing on Bessj and Fisher were low due to nondeterministic solvers but not disabled taint analysis.

For guided random testing with taint analysis enabled, its branch coverage was the same as RT in Table 2 except for Expint where the branch coverage was increased by 22.3%. By closely looking into the test cases that contributed to the improved coverage, we found that these test cases were mutated from the test cases generated by concolic testing to break through corner conditionals. This indicates that concolic testing can guide random testing to cover conditionals that are inside corner ones, as shown in Example 6. The mutation score was lower than RT in Table 2 as Baton is coverage-driven.

For targeted concolic testing with taint analysis enabled, its branch coverage and mutation score were respectively increased by 16.5% and 20.1% when compared to CT in Table 2. This is because random testing relieves some of the burden to cover complex conditionals from concolic testing, and also guides concolic testing to be more targeted and to have less chance to get stuck in constraint solving.

Answer to Q3. Taint analysis makes random testing and concolic testing more targeted, and our guided random testing and targeted concolic testing are synergized to benefit from each other.

4.6 Performance overhead (Q4)

To analyze the performance of each component in Baton, we report the size of conditional execution graph (i.e., the number of nodes), the total time overhead, and the overhead of guided random testing, targeted concolic testing, and strategy analysis in Table 5. Here we report the results for the two benchmarks with large programs.

The total time overhead is nearly proportional to the size of the graph as Baton is guided by systematically exploring the graph. Thus, the size of the graph has a great impact on the performance of Baton. This also explains why we do not unfold loops for balancing scalability and context precision. Moreover, 42%, 30%, and 28% of the time was respectively spent in guided random testing, targeted

Table 5 Performance of each component in Baton

Subject	Size (#)	Total time (s)	Guided random testing (%)	Targeted concolic testing (%)	Strategy analysis (%)
WBS	375	29.3	38	24	38
Raytrace	165	70.5	33	52	15
MinePump	869	90.9	49	18	33
Siena	3787	561.5	21	60	19
NanoXML	395	126.1	39	11	50
Schedule	321	21.8	55	19	26
Schedule2	1381	115.8	55	25	20
Totinfo	40	14.0	23	58	19
PrintTokens2	5267	879.4	57	20	23
Replace	337	46.0	49	16	35
	Average		42	30	28

concolic testing, and strategy analysis, where machine learning, constraint solving, and test execution (together with taint analysis) are respectively the most expensive task in each component. Besides, targeted concolic testing has a higher overhead than guided random testing for Raytrace, Siena, and Totinfo because Raytrace and Totinfo have complex constraints, and Siena has many input-dependent loops.

Answer to Q4. Baton's performance is proportional to the size of the conditional execution graph, which is acceptable given improved branch coverage and mutation score but can be further improved (see more discussions in Subsection 4.7).

4.7 Discussion

The main threat to the validity of the evaluation is that the benchmarks are not very large as Baton uses concolic execution. Baton relieves some burden from concolic testing to random testing, but still partially shares the similar limitations of concolic execution. We used two real-life systems Siena and NanoXML; and further studies are required to generalize the results. We plan to apply concolic execution selectively on code segments where concolic execution is scalable to maximize its effectiveness while improving Baton's scalability.

Besides, n and m were empirically set as the good threshold for the used benchmarks in our evaluation. As a guideline, m can be set to a small value (e.g., 3) to avoid unnecessary effort in random testing, and n can be set to a large value (e.g., 10) to fully utilize the capability of random testing and ensure sufficient training data. We are also investigating the possibility of automatically and dynamically determine n and m by analyzing the coverage progress feedback obtained from existing test case executions.

To improve the performance of Baton, we plan to use incremental machine learning to learn models, reuse the symbolic execution tree in concolic testing instead of constructing it from scratch every time concolic execution is invoked, and perform selective taint analysis only on a small part of test cases. Besides, as the collected symbolic constraints can improve the accuracy of learned models (Subsection 4.2), we are investigating how to use concolic execution to collect symbolic constraints for more conditionals with acceptable overhead. As the proposed approach is general, we are also implementing Baton for programming languages (e.g., C) where the concolic execution engine is more mature with respect to large benchmarks.

Currently, Baton supports primitive types of input variables. Testing programs that process structured inputs is challenging; and Baton treats such inputs as a sequence of chars, which is effective as shown by the fuzzing community [39]. We will improve our mutation strategy to be aware of input structures, following our recent work on grammar-aware fuzzing approach [40, 41].

5 Related work

Here we focus on the most relevant studies, and refer readers to [42–46] for a comprehensive analysis of the state-of-the-art.

5.1 System-level random testing

Random testing [1, 2, 47, 48] often fails to cover all regions of the input domain. To generate diverse test cases, adaptive random testing [49] is proposed, and advanced by many approaches, including fixed size candidate set [3, 30], restricted random testing [31], mirror adaptive random testing [50], adaptive random testing by partition [51], bisection [52], lattice [53], distribution metric [54], and evolutionary adaptive random testing [4]. However, the heavy cost for computing distances of test cases makes it less scalable even for toy programs [18]. To reduce the computation overhead, quasi-random testing [5, 6] applies mathematically developed quasi-random sequences to produce low-discrepant test cases. Shahbazi et al. [19] leveraged the geometric structure of the input domain to achieve speedup. Besides, in the security community, guided fuzzing (e.g., AFL[2)], AFLFast [55], and Steelix [56]) often uses coverage as the feedback to guide random testing. Due to the randomness nature, all these approaches have the difficulty to cover corner conditionals, but Baton integrates concolic testing to handle them.

Taint analysis is used to guide random test generation [57, 58], i.e., to find the inputs that conditionals depend on. Then for partially-covered conditionals, fuzzing [58] or nonlinear optimization technique [57] finds the setting for such inputs to fully cover them. Similarly, we leverage taint analysis to guide random testing, and make concolic testing targeted.

5.2 Unit-level random testing

Guided unit test generation was first introduced by Eclat [59]. Since then, many advances have been made to improve the code coverage. Palulu [60] builds a call sequence model from an example execution, and uses it to create legal method sequences. Randoop [34, 61] can produce diverse test inputs, which use execution feedback to prune illegal or redundant inputs. Yatoh et al. [62] improved Randoop by selecting, adding, and deleting primitive value pools. Ma et al. [63] improved Randoop via static and dynamic program knowledge. Specifically, they reuse constants in the program for the inputs, which becomes less effective when the inputs are involved in complex computations before used in corner conditionals, where concolic execution is helpful. Thus, such a seeding strategy can partially break through corner conditionals but cannot completely substitute the concolic execution in Baton. Further, Baton uses different techniques (e.g., machine learning, taint analysis, and conditional execution graph) to guide random testing than the approach in [63]. To handle the large space of possible method sequences, several techniques are proposed. MSeqGen [64] mines code bases and extracts frequent method sequences that can be used in test generation. Instead of randomly reusing sequences, RecGen [65] recommends sequences that have more relevant methods to the method under test. Similarly, Palus [66] extends Palulu [60] by testing relevant methods together to increase the chance of exploring different program behaviors. Here the relevance of two methods is determined by the fields they both access. Seeker [67] uses program synthesis to construct a method sequence that drives the method under test to reach an uncovered target branch.

These approaches focus on unit testing and handle the problem of how to generate a method sequence to test the method under test. Instead, Baton focuses on system testing, and does not need to handle the method sequence problem. However, it is interesting to investigate how to extend Baton to support unit testing. One simple idea is to apply Baton to each generated unit test, and systematically test the subsystem that each unit test involves. However, because of the huge number of generated unit tests, we will explore how to focus on interesting unit tests that have high potential to lead to faults.

5.3 Search-based testing

Search-based testing [45, 46, 68–70] leverages metaheuristic search techniques to generate test cases that optimize a fitness (e.g., branch coverage or revealed faults) in a limited time budget. Since the early work of Tonella [71], a number of advances have been made. Arcuri and Yao [72], Baresi and Miraz [73], Baars et al. [74], and Lakhotia et al. [75] all targeted a single method sequence during the optimization of one coverage goal. Instead, EvoSuite [35, 76] generates the whole test suites with the aim of satisfying all coverage goals while keeping the total size small, and can achieve high code coverage [77, 78]. EvoSuite is equipped with similar seeding strategies [79, 80] as in [63]. The difference of Baton from such search-based unit testing is the same as guided unit testing (see Subsection 5.2). Several search-based system testing techniques are proposed for domain-specific systems [81, 82]. Arcuri [83] started a search-based

2) American fuzzy lop (2014). http://lcamtuf.coredump.cx/afl/.

system testing project for enterprise systems. It is in the early stage of development, and thus we leave the empirical comparison for future work.

5.4 Concolic testing

Concolic testing [11,12,20,32,84–86] uses dynamic symbolic execution to generate test cases that explore different execution paths of a program. These tools usually use depth-first search as the default exploration strategy. Different exploration strategies have been proposed to choose branches where constraints are negated, e.g., CREST [87], Fitnex [88], and SAGE [89]. CREST [87] applies a control-flow-directed strategy to choose the branch with the minimum distance to an uncovered branch. Fitnex [88] uses a fitness-guided strategy to pick the branch closest to cover a target path. SAGE [89] uses a generational search to negate as many constraints in a path condition as possible. However, they have the difficulty to handle complex conditionals, while Baton relieves some of the burden to random testing. Several advances have been made to improve the scalability of dynamic symbolic execution [90–92]. SMART [90] performs dynamic test generation in a compositional way by generating and reusing function summaries. SAGE [91] uses grammar-based input specification to avoid generating invalid highly-structured inputs. Qi et al. [92] grouped paths that have the same symbolic outputs together so that only one test case is generated for each group, which is much more efficient than the exhaustive path exploration. More recently, Wang et al. [93] proposed to compute the optimal concolic testing strategy (i.e., when to apply concrete execution, when to apply symbolic execution and which program path to apply symbolic execution). They can be leveraged to improve the scalability of Baton as Baton integrates concolic testing.

5.5 Hybrid testing

Majumdar and Sen [13] interleaved random testing with concolic testing, and Driller [14] applied the same idea to binary fuzzing. This is the closest work to Baton. It uses concolic testing to search for an uncovered branch when random testing does not cover new branches; and when an uncovered branch is found by concolic testing, random testing is invoked again. This method leverages concolic testing in a blind way, i.e., many concolic executions are needed to find an uncovered branch; and it is most suitable for testing reactive programs. Instead, we use concolic execution in such a targeted way that only one concolic execution is needed to cover the target conditional. Besides, we use conditional execution graph to distinguish conditional contexts and use machine learning to guide random testing.

For unit testing, Evacon [15] is proposed to integrate evolutionary testing and symbolic execution in a sequential way (without interleaving). Similarly, Galeotti et al. [16] integrated EvoSuite with dynamic symbolic execution. These approaches use symbolic execution in a less efficient way. In addition, Garg et al. [17] proposed to interleave Randoop with concolic testing. When Randoop saturates, a set of uncovered branches in less explored methods are chosen as the target for concolic testing to explore. These approaches provide us with good starting points to extend Baton to support unit testing.

6 Conclusion

We have proposed and implemented a guided hybrid testing approach, named Baton. The key novelty of Baton is that it integrates the knowledge inside test cases and their executions into a conditional execution graph, and uses such knowledge to synergize random testing with concolic testing in a smart way, and it also integrates machine learning and taint analysis to guide random testing and concolic testing effectively. Our evaluation on three benchmarks has demonstrated that Baton can significantly outperform random testing, adaptive random testing, concolic testing, and simple hybrid testing in terms of branch coverage and mutation score. In the future, we plan to (i) improve the scalability of Baton so that it can be used for larger programs, (ii) extend Baton to support unit testing by integrating with existing unit testing techniques to help generate methods so that Baton can support both system and unit testing, and (iii) extend Baton to support the C programming language.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant No. 61802067).

References

1 Loo P, Tsai W. Random testing revisited. Inf Softw Tech, 1988, 30: 402–417

2 Arcuri A, Iqbal M Z, Briand L. Random testing: theoretical results and practical implications. IEEE Trans Softw Eng, 2012, 38: 258–277
3 Chen T Y, Kuo F C, Merkel R G, et al. Adaptive random testing: the ART of test case diversity. J Syst Softw, 2010, 83: 60–66
4 Tappenden A F, Miller J. A novel evolutionary approach for adaptive random testing. IEEE Trans Rel, 2009, 58: 619–633
5 Chen T Y, Merkel R. Quasi-random testing. IEEE Trans Rel, 2007, 56: 562–568
6 Liu H, Chen T Y. Randomized quasi-random testing. IEEE Trans Comput, 2016, 65: 1896–1909
7 Böhme M, Paul S. On the efficiency of automated testing. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014. 632–642
8 Xie T, Marinov D, Schulte W, et al. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2005. 365–381
9 Cadar C, Dunbar D, Engler D. 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, 2008. 209–224
10 Păsăreanu C S, Visser W, Bushnell D, et al. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Autom Softw Eng, 2013, 20: 391–425
11 Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005. 213–223
12 Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005. 263–272
13 Majumdar R, Sen K. Hybrid concolic testing. In: Proceedings of the 29th International Conference on Software Engineering, 2007. 416–426
14 Stephens N, Grosen J, Salls C, et al. Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of Network and Distributed System Security Symposium, 2016
15 Inkumsah K, Xie T. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. 297–306
16 Galeotti J P, Fraser G, Arcuri A. Improving search-based test suite generation with dynamic symbolic execution. In: Proceedings of the 24th International Symposium on Software Reliability Engineering, 2013. 360–369
17 Garg P, Ivancic F, Balakrishnan G, et al. Feedback-directed unit test generation for c/c++ using concolic execution. In: Proceedings of the 35th International Conference on Software Engineering, 2013. 132–141
18 Arcuri A, Briand L. Adaptive random testing: an illusion of effectiveness? In: Proceedings of the 20th International Symposium on Software Testing and Analysis, 2011. 265–275
19 Shahbazi A, Tappenden A F, Miller J. Centroidal voronoi tessellations — a new approach to random testing. IEEE Trans Softw Eng, 2013, 39: 163–183
20 Luckow K, Giannakopoulou D, Howar F, et al. JDart: a dynamic symbolic analysis framework. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2016. 442–459
21 Wang H J, Liu T, Guan X H, et al. Dependence guided symbolic execution. IEEE Trans Softw Eng, 2017, 43: 252–271
22 Hutchins M, Foster H, Goradia T, et al. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Proceedings of the 16th International Conference on Software Engineering, 1994. 191–200
23 Borges M, d'Amorim M, Anand S, et al. Symbolic execution with interval solving and meta-heuristic search. In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation, 2012. 111–120
24 Bell J, Kaiser G. Phosphor: illuminating dynamic data flow in commodity JVMs. In: Proceedings of ACM International Conference on Object Oriented Programming Systems Languages & Applications, 2014. 83–101
25 Frank E, Hall M A, Witten I H. Data Mining: Practical Machine Learning Tools and Techniques. 4th ed. San Francisco: Morgan Kaufmann 2016
26 Hoffmann M R, Janiczak B, Mandrikov E. Eclemma 2.3.3. 2017. http://www.eclemma.org/
27 Just R, Jalali D, Inozemtseva L, et al. Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014. 654–665
28 Ma Y S, Offutt J, Kwon Y R. MuJava: an automated class mutation system. Softw Test Verif Reliab, 2005, 15: 97–133
29 Mayer J, Schneckenburger C. An empirical analysis and comparison of random testing techniques. In: Proceedings of International Symposium on Empirical Software Engineering, 2006. 105–114
30 Chen T, Leung H, Mak I. Adaptive random testing. In: Proceedings of Annual Asian Computing Science Conference, 2005. 320–329
31 Chan K P, Chen T, Towey D. Restricted random testing. In: Proceedings of European Conference on Software Quality, 2002. 321–330
32 Jayaraman K, Harvison D, Ganesh V, et al. JFUZZ: a concolic whitebox fuzzer for java. In: Proceedings of the 1st NASA Formal Methods Symposium, 2009. 121–125
33 de Moura L, Bjørner N. Z3: an efficient smt solver. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008. 337–340
34 Pacheco C, Lahiri S K, Ernst M D, et al. Feedback-directed random test generation. In: Proceedings of International Conference on Software Engineering, 2007. 75–84
35 Fraser G, Arcuri A. Whole test suite generation. IEEE Trans Softw Eng, 2013, 39: 276–291
36 Powers D M W. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. Int J Mach Learn Technol, 2011, 2: 37–63
37 Sheskin D J. Handbook of Parametric and Nonparametric Statistical Procedures. 4th ed. Boca Raton: Chapman & Hall/CRC, 2007
38 Forman G, Cohen I. Learning from little: comparison of classifiers given little training. In: Proceedings of European Conference on Principles of Data Mining and Knowledge Discovery, 2004. 161–172
39 Liang H L, Pei X X, Jia X D, et al. Fuzzing: state of the art. IEEE Trans Rel, 2018, 67: 1199–1218
40 Wang J J, Chen B H, Wei L, et al. Skyfire: data-driven seed generation for fuzzing. In: Proceedings of IEEE Symposium on Security and Privacy, 2017. 579–594
41 Wang J J, Chen B H, Wei L, et al. Superion: grammar-aware greybox fuzzing. In: Proceedings of the 41st International

Conference on Software Engineering, 2019. 724–735
42 Orso A, Rothermel G. Software testing: a research travelogue (2000–2014). In: Proceedings of Future of Software Engineering Proceedings, 2014. 117–132
43 Anand S, Burke E K, Chen T Y, et al. An orchestrated survey of methodologies for automated software test case generation. J Syst Softw, 2013, 86: 1978–2001
44 Păsăreanu C S, Visser W. A survey of new trends in symbolic execution for software testing and analysis. Int J Softw Tools Technol Transfer, 2009, 11: 339–353
45 McMinn P. Search-based software testing: past, present and future. In: Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops, 2011. 153–163
46 McMinn P. Search-based software test data generation: a survey. Softw Test Verif Reliab, 2004, 14: 105–156
47 Hamlet R. Random testing. In: Encyclopedia of Software Engineering. Hoboken: Wiley & Sons, 1994. 970–978
48 Duran J W, Ntafos S C. An evaluation of random testing. IEEE Trans Softw Eng, 1984, 10: 438–444
49 Chen T Y, Tse T H, Yu Y T. Proportional sampling strategy: a compendium and some insights. J Syst Softw, 2001, 58: 65–81
50 Chen T Y, Kuo F C, Merkel R G, et al. Mirror adaptive random testing. In: Proceedings of the 3rd International Conference on Quality Software, 2003. 4–11
51 Chen T Y, Merkel R, Wong P K, et al. Adaptive random testing through dynamic partitioning. In: Proceedings of the 4th International Conference on Quality Software, 2004. 79–86
52 Mayer J. Adaptive random testing by bisection and localization. In: Proceedings of International Workshop on Formal Approaches to Software Testing, 2006. 72–86
53 Mayer J. Lattice-based adaptive random testing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005. 333–336
54 Chen T Y, Kuo F C, Liu H. Adaptive random testing based on distribution metrics. J Syst Softw, 2009, 82: 1419–1433
55 Bohme M, Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. IEEE Trans Softw Eng, 2019, 45: 489–506
56 Li Y K, Chen B H, Chandramohan M, et al. Steelix: program-state based binary fuzzing. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, 2017. 627–637
57 Leek T R, Baker G Z, Brown R E, et al. Coverage Maximization Using Dynamic Taint Tracing. Massachusetts Inst Of Tech Lexington Lincoln Lab Technical Report, 2007
58 Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing. In: Proceedings of the 31st International Conference on Software Engineering, 2009. 474–484
59 Pacheco C, Ernst M D. Eclat: automatic generation and classification of test inputs. In: Proceedings of European Conference on Object-Oriented Programming, 2005. 504–527
60 Artzi S, Ernst M D, Zun A K, et al. Finding the needles in the haystack: generating legal test inputs for object-oriented programs. In: Proceedings of the 1st Workshop on Model-Based Testing for Object-Oriented Systems (M-TOOS), 2006
61 Pacheco C, Lahiri S K, Ball T. Finding errors in .Net with feedback-directed random testing. In: Proceedings of International Symposium on Software Testing and Analysis, 2008. 87–96
62 Yatoh K, Sakamoto K, Ishikawa F, et al. Feedback-controlled random test generation. In: Proceedings of International Symposium on Software Testing and Analysis, 2015. 316–326
63 Ma L, Artho C, Zhang C, et al. GRT: program-analysis-guided random testing. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, 2015. 212–223
64 Thummalapenta S, Xie T, Tillmann N, et al. MSeqGen: object-oriented unit-test generation via mining source code. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009. 193–202
65 Zheng W J, Zhang Q R, Lyu M, et al. Random unit-test generation with MUT-aware sequence recommendation. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering, 2010. 293–296
66 Zhang S, Saff D, Bu Y Y, et al. Combined static and dynamic automated test generation. In: Proceedings of International Symposium on Software Testing and Analysis, 2011. 353–363
67 Thummalapenta S, Xie T, Tillmann N, et al. Synthesizing method sequences for high-coverage testing. SIGPLAN Not, 2011, 46: 189–206
68 Ali S, Briand L C, Hemmati H, et al. A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans Softw Eng, 2010, 36: 742–762
69 Harman M, McMinn P. A theoretical and empirical study of search-based testing: local, global, and hybrid search. IEEE Trans Softw Eng, 2010, 36: 226–247
70 Harman M, Jia Y, Zhang Y Y. Achievements, open problems and challenges for search based software testing. In: Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST), 2015. 1–12
71 Tonella P. Evolutionary testing of classes. In: Proceedings of ACM/SIGSOFT International Symposium on Software Testing and Analysis, 2004. 119–128
72 Arcuri A, Yao X. Search based software testing of object-oriented containers. Inf Sci, 2008, 178: 3075–3095
73 Baresi L, Miraz M. Testful: automatic unit-test generation for java classes. In: Proceedings of the 32nd International Conference on Software Engineering, 2010. 281–284
74 Baars A, Harman M, Hassoun Y, et al. Symbolic search-based testing. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, 2011. 53–62
75 Lakhotia K, Harman M, Gross H. AUSTIN: an open source tool for search based software testing of C programs. Inf Softw Tech, 2013, 55: 112–125
76 Fraser G, Arcuri A. Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011. 416–419
77 Fraser G, Arcuri A. Sound empirical evidence in software testing. In: Proceedings of the 34th International Conference on Software Engineering, 2012. 178–188
78 Fraser G, Arcuri A. A large-scale evaluation of automated unit test generation using evosuite. ACM Trans Softw Eng Methodol, 2014, 24: 1–42
79 Fraser G, Arcuri A. The seed is strong: seeding strategies in search-based software testing. In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation, 2012. 121–130
80 Rojas J M, Fraser G, Arcuri A. Seeding strategies in search-based unit test generation. Softw Test Verif Reliab, 2016, 26:

366–401
81 Gross F, Fraser G, Zeller A. Search-based system testing: high coverage, no false alarms. In: Proceedings of International Symposium on Software Testing and Analysis, 2012. 67–77
82 Abdessalem R B, Nejati S, Briand L C, et al. Testing advanced driver assistance systems using multi-objective search and neural networks. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016. 63–74
83 Arcuri A. Evomaster: evolutionary multi-context automated system test generation. In: Proceedings of the 11th International Conference on Software Testing, Verification and Validation, 2018. 394–397
84 Sen K, Agha G. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: Proceedings of International Conference on Computer Aided Verification, 2006. 419–423
85 Cadar C, Ganesh V, Pawlowski P M, et al. EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, 2006. 322–335
86 Tillmann N, de Halleux J. Pex: white box test generation for .Net. In: Proceedings of International Conference on Tests and Proofs, 2008. 134–153
87 Burnim J, Sen K. Heuristics for scalable dynamic test generation. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. 443–446
88 Xie T, Tillmann N, de Halleux J, et al. Fitness-guided path exploration in dynamic symbolic execution. In: Proceedings of IEEE/IFIP International Conference on Dependable Systems & Networks, 2009. 359–368
89 Godefroid P, Levin M Y, Molnar D A. Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security (NDSS) Symposium, 2008
90 Godefroid P. Compositional dynamic test generation. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2007. 47–54
91 Godefroid P, Kiezun A, Levin M Y. Grammar-based whitebox fuzzing. SIGPLAN Not, 2008, 43: 206–215
92 Qi D W, Nguyen H D, Roychoudhury A. Path exploration based on symbolic output. In: Proceedings of Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2011. 278–288
93 Wang X Y, Sun J, Chen Z B, et al. Towards optimal concolic testing. In: Proceedings of the 40th International Conference on Software Engineering, 2018. 291–302