Gray-Box Fuzzing via Gradient Descent and Boolean Expression Coverage*

Martin Jonáš, Jan Strejček, Marek Trtík(✉), and Lukáš Urban

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{martin.jonas,strejcek,trtikm,492717}@mail.muni.cz

Abstract. We present a gray-box fuzzing approach based on several new ideas. While standard gray-box fuzzing aims to cover all branches of the input program, our approach primarily aims to cover both results of each Boolean expression. To achieve this goal, we track the distances to flipping these results and we dynamically detect the input bytes that influence the distance. Then we use this information to efficiently flip the results. More precisely, we apply gradient descent on the detected bytes or we create new inputs by using detected bytes from different inputs.
We implemented our approach in a tool called Fizzer. An evaluation on the benchmarks of Test-Comp 2023 shows that Fizzer is fully competitive with the winning tools of the competition, which use advanced formal methods like symbolic execution or bounded model checking, usually in combination with fuzzing.

1 Introduction

Fuzzing is a technique for automated generation of test inputs for a given program. The goal of fuzzing is to generate tests with high code coverage and to quickly detect bugs in the code. We distinguish three basic kinds of fuzzing based on their use of the given program. *Black-box fuzzing* [18] only runs the given program on various inputs and observes the outputs. *Gray-box fuzzing* [18] first instruments the program to get some information about performed executions. The instrumented code typically tracks the information about the basic blocks visited during the execution. While black-box and gray-box fuzzing rely on dynamic analysis of the original or instrumented code, *white-box fuzzing* [18] combines dynamic analysis with some static analysis of the code, typically concolic execution, symbolic execution, or bounded model checking.

Black-box fuzzers have only limited efficiency due to the lack of information. Gray-box fuzzers and white-box fuzzers proved to be very efficient and they are routinely applied in software industry. For example, the gray-box fuzzer AFL [27] discovered dozens of bugs in many recognized open-source projects and the white-box fuzzer Sage [11] is intensively used in Microsoft.

The standard approach of successful gray-box fuzzers is to collect only a very limited information about each program execution and to quickly perform

* This work has been supported by the Czech Science Foundation grant GA23-06506S.

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 90–109, 2024.
https://doi.org/10.1007/978-3-031-57256-2_5

as many executions as possible. In this paper we suggest an approach that gathers slightly more information about program executions and uses it to select uncovered parts of the code and make more targeted attempts to cover it. We can illustrate some ideas of our approach on a simple example. Consider a program that contains a branching statement `if (x > 42)` and assume that some program execution passed its `true` branch. During this program execution, we saved the value of `x - 42` to know the *distance* to entering the `false` branch. When we decide to cover the `false` branch, we first repeatedly execute the program on modified inputs to detect the bytes of the input that have some influence on the distance value. This is called a *sensitivity analysis* and the detected bytes are called *sensitive*. We then propose two analyses that use the sensitive bytes to cover the uncovered branch. One analysis performs a dynamic *gradient descent* on the sensitive bytes with the aim to minimize the absolute value of the distance and to enter the `false` branch. Alternatively, if we already know another input that entered the `false` branch of this statement in a different calling context, we can try to use the value of its sensitive bytes instead of the sensitive bytes of the current input. This analysis is called *byteshare analysis*. Now consider a slightly different program where the branching statement has the form `if (res)` where `res` is a Boolean variable assigned before by `res = x > 42`. Clearly, we want to track the distance to changing the value of `res`. Hence, we in fact do not track distances for branching conditions, but the distances for values of atomic Boolean expressions. Roughly speaking, our approach aims to generate tests such that each atomic Boolean expression in each calling context is evaluated to `true` and to `false` in some program executions. Our fuzzing approach tracks its progress with the use of *atomic Boolean execution tree* and we talk about *Boolean expression coverage*.

The following section introduces the basic terminology used in the paper and states our assumptions on the analysed programs. Section 3 then describes the basic concepts of our approach, in particular the Boolean expression coverage, the information we collect from each program execution and how we obtain this information, the atomic Boolean execution tree, and the fuzzing algorithm. This algorithm iteratively tries to close vertices of the tree by generating inputs in which each of the vertices evaluates both to `true` and to `false` in order to either increase the Boolean expression coverage or to discover new parts of the tree. These inputs are generated by sensitivity analysis, byteshare analysis, and gradient descent analysis presented in Section 4. The selection strategy of the vertex to be closed is briefly explained in Section 5. Note that the page limit does not allow describing all the technical details of the approach. They can be found in the corresponding technical report [15].

We have implemented the presented fuzzing approach in a tool called Fizzer. The architecture and some implementation aspects of the tool are described in Section 6. Further, we have run Fizzer on all benchmarks of the *Cover-Branches* category of the *Competition on Software Testing (Test-Comp) 2023* [5]. We evaluated the tests generated by Fizzer using the competition infrastructure which measures the achieved branch coverage. The results presented in Section 7 show

that our tool is competitive with the top-ranking tools of Test-Comp 2023, namely FuSeBMC [2], VeriFuzz [21], and CoVeriTest [6]. Note that our tool is a pure gray-box fuzzer while FuSeBMC and VeriFuzz combine dynamic analysis with static analyses like symbolic execution and bounded model checking. CoVeriTest fully relies on static methods like predicate analysis with the CEGAR loop and value analysis. Finally, Section 8 discusses some related work and Section 9 sums up the presented results and outlines future work.

2 Preliminaries

The ideas presented in this paper can be adopted for various kinds of programs. For ease of presentation, here we consider sequential C programs that get input only via functions `nondet_char()`, `nondet_int()`, and `nondet_float()` which return values of the corresponding type.

For simplicity, we assume that these are the only types that can be read from the input and we define the set $\mathit{InputTypes} = \{\texttt{char}, \texttt{int}, \texttt{float}\}$. We define the set of typed values $\mathit{TypedValues} = \{(v, t) \mid t \in \mathit{InputTypes}, v \text{ is a value of type } t\}$ and denote the pairs $(v, t) \in \mathit{TypedValues}$ as $v : t$, e.g., $3 : \texttt{int}$ is the value 3 of type `int`. We also work with untyped inputs, which are arbitrary finite sequences of bits 0 and 1. Untyped inputs are denoted by a standard language-theoretic notation, e.g., 1^{12} is a sequence of 12 elements 1.

An expression occurring in a program is called an *atomic Boolean expression* (ABE) if it has type `bool` and it is not a variable, not a call of a function whose definition is a part of the program, and not a result of applying logical operators, i.e., conjunction, disjunction, and negation. For example, the expression `(x > 3) && foo(x,y) && cond`, where `foo` is a function defined in the program and `cond` is a variable, contains only one ABE `x > 3`. By ABE we always mean a particular occurrence of the expression in the program.

We assume that the control flow is fully determined by the values of ABEs. This property may not hold for programs with `switch` statements, function calls via input-dependent function pointers, etc. However, such programs can be transformed into equivalent ones satisfying our assumption.

By a *calling context* we mean the sequence of function calls that are currently being evaluated. The outermost function call is the first element of the sequence and the last one is the function whose body is executed at the moment. In other words, the calling context roughly corresponds to the call stack.

We sometimes denote a sequence $x_1 x_2 \dots x_n$ as $\langle x_1, x_2, \dots, x_n \rangle$ or $\langle x_i \rangle_{1 \leq i \leq n}$.

3 Overview of Our Fuzzing Approach

This section provides an overview of the key concepts that are used in our fuzzing algorithm and presents the high-level view of the algorithm. The key heuristics for input generation are explained later in Sections 4 and 5.

```
void main() {
  int x = nondet_int();
  if (x < 42) {
    // branch 1
  } else {
    // branch 2
  }
}
```

(a) Trivial case.

```
void main() {
  int x = nondet_int();
  bool res1 = x < 42;
  x++;
  bool res2 = x < 42;
  if (res1 || res2) {
    // branch 1
  } else {
    // branch 2
  }
}
```

(b) Depends on a non-local comparison.

```
bool compare(int v) {
  return v < 42;
}

void main() {
  int x = nondet_int();
  bool res1 = compare(x);
  x++;
  bool res2 = compare(x);
  if (res1 || res2) {
    // branch 1
  } else {
    // branch 2
  }
}
```

(c) Depends on a comparison coming from a different scope.

Listing 1.1: Example C codes showing that the values that influence which branch is taken can be both lexically far away from the branching statement and can be behind several layers of indirection.

3.1 Branch Coverage via Boolean Expression Coverage

The main idea of the proposed approach is to assign to each executed branching statement a metric called *distance* reflecting how far the current program state is from evaluating the branching expression to the opposite Boolean value. Thanks to this metric, we can use gradient descent to generate inputs that either flip the Boolean value or are close enough to the flipping point so that the actual flip can be achieved by small mutations of the input.

It is easy to define the distance for branchings like `if (x > 42)`: we set the distance to $x-42$ and minimize the absolute value $|x-42|$ to get close to the point where the result of the branching expression changes. However, as Listing 1.1 shows, the situation can be far more complex. The comparison does not have to occur in the branching expression itself, but it can be precomputed earlier in the program, it can come from a function call or be read from an array, etc.

We sidestep this issue by assigning the distances to atomic Boolean expressions and trying to flip their values rather than doing the same for branching expressions. In other words, we approach the goal of generating tests with maximal branch coverage indirectly by maximizing *Boolean expression coverage*. Intuitively, we try to generate a set of inputs such that every *atomic Boolean expression* evaluates to `true` on some input and to `false` on some input. In fact, we want to generate inputs leading to both Boolean values of each ABE in *each possible calling context*. The importance of the calling context is illustrated on the ABE `v < 42` in the code of Listing 1.1c: we clearly want to distinguish the case when the value of `v < 42` is used to set the value of `res1` from the case when it is used to set the value of `res2`. The precise goal of our approach will be formulated later using the terms *atomic Boolean execution tree* and *covered vertex* of the tree introduced in Definitions 1 and 2, respectively.

Every time an ABE e is evaluated, its *distance* is computed by the expression

$$dist(e) = \begin{cases} value(l) - value(r), & \text{if } e = l \bowtie r \text{ where } \bowtie \in \{=, \neq, <, \leq, >, \geq\}, \\ value(e), & \text{otherwise.} \end{cases}$$

In the first case, the $value(l)$ and $value(r)$ refer to the numerical values of l and r, respectively, before the evaluation of e. In the second case, $value(e)$ is defined as 1 if e evaluates to `true` and it is defined as 0 if e evaluates to `false`.

Note that the branch coverage and the atomic Boolean expression coverage do not precisely match. For example, we can achieve the full branch coverage of the code in Listing 1.1b by two tests; with input values 40 : `int` and 41 : `int`. However, the `true` branch of the first ABE `x < 42` is not covered by either of these tests. Nevertheless, our experimental evaluation shows that maximizing the atomic Boolean expression coverage also leads to test inputs with high branch coverage.

3.2 Instrumentation and Execution

From each program execution, our approach needs to get the sequence of evaluated ABEs including their calling contexts, their Boolean values, and their distances. To obtain this information, the program is instrumented with the following functions:

- To track the calling context, we assign a unique identifier *id* to each function call (except `nondet_*` function calls) and insert `__instr_call(`*id*`)` before the call and `__instr_return()` after the call. The inserted function calls maintain the current stack of open function calls.
- To track all evaluated ABEs and their values, distances, and calling contexts, we assign a unique identifier *id* to each ABE e and insert the call `__instr_abe(`*id*`,` e`,` $dist(e)$`)` before the ABE. The calling context is internally retrieved from the tracked stack of open function calls.

Listing 1.2 provides the instrumented programs from Listings 1.1b and 1.1c.

Besides the inserted function calls, we also alter the functions `nondet_`*type*`()` to collect the information about the values and types read from the input stream and when they were read.

In the following, we assume that there exists a function `execute`$(P', input)$ that gets an instrumented program P' and an untyped input $input \in \{0,1\}^*$ and returns the *trace* of the execution of P' on $input.0^\omega$, i.e., *input* extended with infinitely many zero bits. The trace is a pair $(usedInput, \pi)$ where

- *usedInput* is the sequence of *TypedValues* that were read by the program P' during the execution.
- π is the sequence $\langle (e_i, c_i, r_i, d_i, n_i) \rangle_{1 \leq i \leq k}$ of tuples, where each tuple represents one evaluation of an ABE: e_i is the evaluated ABE, c_i is the calling context in which it was evaluated, r_i is the result of the evaluation, d_i is the corresponding value of $dist(e_i)$, and n_i is the number of bytes of the input that have been read before the evaluation.

```
void main() {
  int x = nondet_int();
  __instr_abe(1, x < 42, x - 42);
  bool res1 = x < 42;
  x++;
  __instr_abe(2, x < 42, x - 42);
  bool res2 = x < 42;
  if (res1 || res2) {
    // branch 1
  } else {
    // branch 2
  }
}
```

(a) Instrumentation of Listing 1.1b

```
bool compare(int v) {
  __instr_abe(1, v < 42, v - 42);
  return v < 42;
}

void main() {
  int x = nondet_int();
  __instr_call(1);
  bool res1 = compare(x);
  __instr_return();
  x++;
  __instr_call(2);
  bool res2 = compare(x);
  __instr_return();
  if (res1 || res2) {
    // branch 1
  } else {
    // branch 2
  }
}
```

(b) Instrumentation of Listing 1.1c

Listing 1.2: Instrumented programs from Listings 1.1b and 1.1c

Note that the trace is always finite as P' is executed with some limits on the number of evaluated ABEs.

Example 1. Let P' be the instrumented program from Listing 1.2b. The function `execute`$(P', 0^{32})$ returns the trace $(\langle 0 : \texttt{int}\rangle, \pi)$, where

$$\pi = \big\langle (\texttt{v < 42}, \langle 1\rangle, \texttt{true}, -42, 4), (\texttt{v < 42}, \langle 2\rangle, \texttt{true}, -41, 4)\big\rangle.$$

In other words, the execution read only a single `int` of value 0 from the input and these 4 bytes were read before the first ABE evaluation. Further, the ABE `v < 42` with identifier 1 (for readability denoted directly by the expression) has been evaluated twice: once in the calling context $\langle 1\rangle$, value `true`, and distance -42 and later with the calling context $\langle 2\rangle$, value `true`, and distance -41.

3.3 Atomic Boolean Execution Tree

Each execution trace $(\mathit{usedInput}, \langle (e_i, c_i, r_i, d_i, n_i)\rangle_{1\le i\le k})$ determines the sequence $r_1 r_2 \dots r_k$ of ABE values. Our fuzzing approach tracks the information about all such sequences seen so far by maintaining an *atomic Boolean execution tree*.

Definition 1 (atomic Boolean execution tree, ABET). *An* atomic Boolean execution tree *(*ABET*) is a nonempty prefix-closed finite set* $T \subseteq \{\texttt{true}, \texttt{false}\}^*$. *Elements of* T *are vertices,* ε *is the root, and elements* $v.\texttt{true}, v.\texttt{false}$ *are children of* v. *We assume that each vertex is either a leaf or it has two children, i.e., for each* $v \in T$ *it holds* $v.\texttt{true} \in T \iff v.\texttt{false} \in T$.

Our method starts with the tree $T = \{\varepsilon\}$. Whenever we obtain a trace $(\mathit{usedInput}, \langle (e_i, c_i, r_i, d_i, n_i)\rangle_{1\le i\le k})$, we update T to contain the sequence $r_1 \dots r_k$

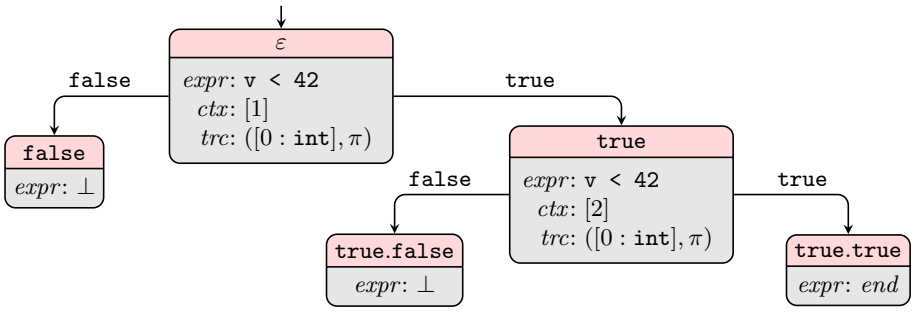

Fig. 1: An example of an ABET.

and all its prefixes. Further, with each newly added vertex we also add its sibling. We say that the trace *visits* a vertex v if v is a prefix of $r_1 r_2 \dots r_k$.

As we mentioned in the preliminaries, we assume that the next evaluated ABE of each program is fully determined by the values of ABEs evaluated before it. This means that each inner vertex $v \in T$ determines the corresponding ABE and its calling context. The ABE and the calling context corresponding to v are denoted by $expr(v)$ and $ctx(v)$. We extend the notation $expr(v)$ also to leaves. We set $expr(v) = end$ if we have seen a trace with the sequence v of ABE values. If this is not the case and v is in T only because of its sibling (or as the only node in the initial tree $\{\varepsilon\}$), we set $expr(v) = \bot$. Note that a leaf v with $expr(v) = \bot$ can become a leaf with $expr(v) = end$ or even an inner node if we later obtain a trace that visits v. Similarly, a leaf v with $expr(v) = end$ can become an inner node. This happens for example when v originally represents a trace that ends with an error (e.g., division by zero) and later we found a longer trace visiting v that avoids the error.

Finally, to each inner vertex $v \in T$ we associate some trace that visits it. The trace is denoted as $trc(v)$.

An example of an ABET can be found in Figure 1. It represents the tree for the instrumented program from Listing 1.2b after obtaining the first trace $(\langle 0 : \texttt{int} \rangle, \pi)$ given in Example 1.

Definition 2 (Covered vertex). *An inner vertex $v \in T$ is said to be* covered *if there are inner vertices $v_t, v_f \in T$ satisfying $expr(v) = expr(v_t) = expr(v_f)$, $ctx(v) = ctx(v_t) = ctx(v_f)$, $expr(v_t.\texttt{true}) \neq \bot$, and $expr(v_f.\texttt{false}) \neq \bot$. An inner vertex that is not covered is called* uncovered.

Definition 3 (Open/closed vertex). *An inner vertex $v \in T$ is said to be* open *if $expr(v.\texttt{true}) = \bot$ or $expr(v.\texttt{false}) = \bot$. An inner vertex that is not open is called* closed.

3.4 Fuzzing Algorithm

A high-level description of the fuzzing algorithm is given in Algorithm 1. The algorithm starts with instrumentation of the given program P (line 1) and ini-

Algorithm 1 Fuzzing algorithm

1: create instrumented program P' from P (see Section 3.2)
2: $T \leftarrow \{\varepsilon\}$
3: $(usedInput, \pi) \leftarrow \texttt{execute}(P', \varepsilon)$
4: $\texttt{processTrace}(usedInput, \pi)$
5: **while** some inner vertex of T is not covered **do**
6: select an unprocessed open vertex v from T (see Section 5)
7: **if** no v is selected **then** end test generation
8: try to close v in T using an input generation analysis (see Section 4)

tialization of the ABET T (line 2). Then it executes the instrumented program on the stream of zero bits (line 3) to obtain an initial trace $(usedInput, \pi)$ where π is of the form $\langle (e_i, c_i, r_i, d_i, n_i) \rangle_{1 \leq i \leq k}$.

On line 4, the trace is processed by $\texttt{processTrace}(usedInput, \pi)$. This function updates T with the sequence $r_1 r_2 \dots r_k$ as described in Section 3.3. For each inner vertex $v \in T$ visited by the current trace and not visited by any trace before, we set $trc(v)$ to the current trace. Further, for each vertex $v \in T$ visited by the current trace and another trace before, we compute the value $\sum_{i=1}^{|v|+1} d_i^2$ and if it is smaller than the corresponding value for $trc(v)$, we set $trc(v)$ to the current trace. Our practical experiments showed that keeping the trace with the smaller sum of squares of d_i leads to better results than minimizing only the current distance $|d_{|v|+1}|$. Finally, the function $\texttt{processTrace}(usedInput, \pi)$ saves *usedInput* to the output test suite if it is in $trc(v)$ of some vertex v at this moment. Otherwise, the trace is completely discarded.

The main fuzzing loop (line 5) iterates until all vertices in T are covered. In each iteration, we select an unprocessed open vertex $v \in T$ (line 6). A vertex is *processed* if it has been analyzed by all input generation analyses. If we fail to select v, the fuzzing algorithm terminates (line 7). Otherwise, we try to close v by some input generation analysis (line 8). The selection process and the input generation analyses are described in Sections 5 and 4, respectively.

4 Input Generation

We propose three methods to generate new inputs with the aim to close the selected vertex: sensitivity analysis, byteshare analysis, and gradient descent. When a vertex is selected, we execute the first of these analyses that has not been executed yet for the vertex. The order is important, as byteshare and gradient descent analyses need the information about sensitive bytes, and byteshare analysis is significantly cheaper than the gradient descent analysis.

In all the analyses, v is the vertex we want to close, we assume without loss of generality that $expr(v.\texttt{true}) = \bot$, and we define $l = |v| + 1$, i.e., the depth of v. The goal of all analyses is to generate an input for which the resulting trace visits v and continues to $v.\texttt{true}$. In all the analyses, $trc(v) = (usedInput, \pi)$ denotes the current trace assigned to the vertex v, with the typed values read

by the trace $usedInput = \langle in_i : t_i \rangle_{1 \leq i \leq n}$ and the sequence of ABE evaluations $\pi = \langle (e_i, c_i, r_i, d_i, n_i) \rangle_{1 \leq i \leq k}$. Moreover, whenever any of the analyses executes P', the resulting execution trace is processed by `processTrace` function.

4.1 Sensitivity Analysis

The goal of the analysis is twofold. First, it detects so-called *sensitive bytes* of vertex v, denoted as $sbytes(v)$. Let us denote as $b_{i,j}$ the j-th byte in the i-th typed value in_i. We check whether $b_{i,j}$ is sensitive by mutating each bit of the byte $b_{i,j}$ separately and executing the program P' on each one-bit mutation. If the resulting trace with $\pi' = \langle (e'_i, c'_i, r'_i, d'_i, n'_i) \rangle_{1 \leq i \leq k'}$ still visits v and the value of the distance function in the node v changes, i.e., $d'_l \neq d_l$, the whole byte is considered *sensitive* and is added to $sbytes(v)$. We also try changing the whole value in_i to several selected special values, e.g., the smallest and the greatest value of the type t_i and special floating-point values, if t_i is `float`.

Second, during the computation of sensitive bytes, we also extend the tree with each executed trace. The sensitivity analysis therefore also effectively works as a local neighborhood search around the previous input of the vertex v.

Observe that when computing sensitive bytes of the vertex v, we can simultaneously use the resulting traces to determine the sensitive bytes of all predecessors of v. We use this observation as an optimization in the implementation to reduce the number of sensitivity analysis executions.

4.2 Byteshare Analysis

Let u be an inner vertex of the current tree T with the same ABE as v (the contexts may differ), with a non-empty set of the sensitive bytes, and whose successor u.`true` is not a leaf. For each such vertex u, the analysis combines inputs from $trc(u.\texttt{true})$ and $trc(v)$ into a new input. More precisely, the new input is the same as $trc(v)$, but for each $j \in \{1, 2, \ldots, \min(|sbytes(v)|, |sbytes(u)|)\}$, we replace the value of the j-th sensitive byte of v by the value of the j-th sensitive byte of u in $trc(u.\texttt{true})$. The idea behind this construction is that we keep the new input similar to the original input of $trc(v)$ so that the execution trace will likely visit v, but we replace the sensitive bytes of v by those of $u.\texttt{true}$, which might steer the execution to the desired child.

Note that $sbytes(v)$ and $sbytes(u)$ may be completely different bytes. The size of the sets may also differ. Since we lack information for building a mapping between $sbytes(v)$ and $sbytes(u)$, we simply map the bytes based on their order.

4.3 Gradient Descent with Multi-sampling and Locking

We extend the notion of sensitivity to the typed inputs. An element of the sequence *usedInputs* is called *sensitive* in v if it contains at least one byte sensitive in v. The gradient descent analysis tries to minimize the absolute value of the distance for v by changing only the sensitive typed inputs of the vertex v. We

Algorithm 2 Gradient descent for vertex v from seed $\boldsymbol{x}$ and distance $f(\boldsymbol{x})$

1: **while** v is open and the number of steps is below the predefined bound **do**
2: **for all** $i \in \{1, \ldots, m\}$ **do**
3: compute $\nabla_i f(\boldsymbol{x})$ as $\frac{|\texttt{ComputeDistance}(x_1,\ldots,x_{i-1},x_i+\Delta x_i,x_{i+1},\ldots,x_m)|-|f(\boldsymbol{x})|}{\Delta x_i}$
4: lock each $\nabla_i f(\boldsymbol{x})$ which is not finite
5: **while** $||\nabla f(\boldsymbol{x})||^2$ is finite and non-zero **do**
6: $\lambda \leftarrow |f(\boldsymbol{x})| \;/\; ||\nabla f(\boldsymbol{x})||^2$
7: **if** λ is zero or not finite **then return**
8: $V' \leftarrow \emptyset$
9: **for all** $e \in \{0, -1, 1, -2, 2, -3, 3\}$ **do**
10: $\boldsymbol{x}' \leftarrow \boldsymbol{x} - 10^e \lambda \nabla f(\boldsymbol{x})$
11: $V' \leftarrow V' \cup \{(\boldsymbol{x}', \texttt{ComputeDistance}(\boldsymbol{x}'))\}$
12: let $(\boldsymbol{x}', f(\boldsymbol{x}')) \in V'$ be the pair with the smallest finite $|f(\boldsymbol{x}')|$
13: **if** $|f(\boldsymbol{x}')| < |f(\boldsymbol{x})|$ **then**
14: $\boldsymbol{x} \leftarrow \boldsymbol{x}'$, $f(\boldsymbol{x}) \leftarrow f(\boldsymbol{x}')$
15: **break**
16: **else**
17: lock all extreme coordinates $\nabla_i f(\boldsymbol{x})$
18: **if** no coordiate was locked **then return**

fix the values of the inputs that were not identified as sensitive as they likely do not influence the value of the distance. In particular, we minimize the function $f(\boldsymbol{x})$ that receives an input vector of m values that correspond to sensitive inputs of the vertex v. The value of the function $f(\boldsymbol{x})$ is computed by a function $\texttt{ComputeDistance}(\boldsymbol{x})$ that:

1. Creates the input sequence $input'$ by replacing the sensitive inputs of the original input from $trc(v)$ by the values specified in $\boldsymbol{x}$.
2. Executes the program on $input'$ and obtains the trace $(usedInput', \pi')$, where $\pi' = \langle (e'_i, c'_i, r'_i, d'_i, n'_i) \rangle_{1 \le i \le k'}$.
3. If the trace π' does not visit v, returns ∞.
4. Otherwise returns the obtained distance value at the vertex v, i.e., d'_l.

The search for the desired values of $\boldsymbol{x}$ is motivated by the following idea. If $\boldsymbol{x}$ is chosen from a small neighborhood around the global minimum of $|f(\boldsymbol{x})|$, the value $f(\boldsymbol{x})$ has roughly the same chance of being positive as negative. I.e., there is roughly the same chance of $expr(v)$ being evaluated to `true` as `false`. Therefore, we repeatedly run the gradient descent from randomly chosen seeds $\boldsymbol{x}$ to approach towards the minimum. Along the way, we perform sampling in the descent direction. This sampling also helps escaping from local minima by trying more values of the function $f(\boldsymbol{x})$.

Our gradient descent starting from one random seed $\boldsymbol{x}$ is formally described in Algorithm 2. We repeatedly perform gradient descent steps from the initial seed $\boldsymbol{x}$ until we generate an input that closes the open vertex v or reach the predefined bound on gradient descent steps. In the loop at line 2, we numerically compute coordinates $\nabla_i f(\boldsymbol{x})$, one for each variable x_i, of the gradient vector

$\nabla f(\boldsymbol{x})$. The coordinates are computed using forward differences, where $\Delta x_i > 0$ is the smallest change of that variable. Since the algorithm works only with finite values, all non-finite coordinates $\nabla_i f(\boldsymbol{x})$ are locked, i.e., they are set to zero and we do not move in these coordinates in the gradient step.

The loop at line 5 performs a single gradient step. It first computes the value of learning rate λ at line 6, which has the property that the *linear approximation* of the function f at $\boldsymbol{x}$ is zero at the input $\boldsymbol{x} - \lambda \nabla f(\boldsymbol{x})$. Next we compute a set V' of samples $\boldsymbol{x}'$ (see the loop at line 9), each representing a candidate for the gradient step. Observe that the samples are separated by multipliers 10^e ranging over several orders of magnitude. These are the samples we mentioned earlier, which can both explore the small neighborhood of the global minimum and escape from local minima. Only the sample $\boldsymbol{x}'$ with the smallest $|f(\boldsymbol{x}')|$ is considered in the gradient step (see line 12). If none of the samples decreases the value of the function, we are stuck in a local minimum and try to escape it by locking more coordinates of the gradient. Namely, we identify and lock coordinates with high absolute values compared to others as they dominate the descent direction. By their locking, we can dramatically change the descent direction and potentially move towards the global minimum. If all coordinates are locked, i.e., set to zero, $||\nabla f(\boldsymbol{x})||^2 = \sum_i (\nabla_i f(\boldsymbol{x}))^2$ will be zero and the gradient descent terminates.

The gradient descent algorithm is repeatedly called with randomly chosen seed inputs $\boldsymbol{x}$ and the starting distance $f(\boldsymbol{x}) = \texttt{ComputeDistance}(\boldsymbol{x})$, until the target vertex is closed[1] or we exceed the predefined bound on the number of seeds to try. We skip all the seeds $\boldsymbol{x}$ for which $\texttt{ComputeDistance}(\boldsymbol{x})$ is infinite. More details of the algorithm can be found in the technical report [15].

5 Target Vertex Selection

We now briefly describe how we select vertices that are targeted by the analyses from the previous section. First, the heuristic tries to select a suitable *uncovered* vertex that has not been processed yet. Second, if all *uncovered* vertices have been processed, it means that none of the analyses was able to cover them. In this case, we try to select an *open* unprocessed vertex and try to close it. The detailed description of the selection process is available in the technical report [15].

5.1 Selecting an Uncovered Vertex

Primarily, we want to target uncovered vertices. Before that, we want to explore program executions with diverse numbers of loop iterations. To this end, we would like to identify all *loop head* vertices in the ABET, which can be expensive. Therefore, we perform loop head detection lazily on the fly. We maintain a worklist of loop heads H and if it is not empty, we remove its random vertex

[1] In fact, Algorithm 2 is immediately terminated when the target vertex is closed by any execution of `ComputeDistance`.

and select it as the target. Only if the worklist H is empty, we select a suitable vertex v in the tree based on *vertex selection heuristics* and detect loop heads on the path to the vertex v. If there are loop heads on the path to v, we put some of them to H based on the *loop head selection heuristics* and randomly take one of them as the target vertex. If there are no loop heads on the path to v or the loop heads on the path to v have been processed, we select v itself as the target vertex. We now describe the heuristics that we use for selection the suitable vertex v and for selection of loop heads on the path to v.

Vertex Selection Heuristics The selection relies on the classification of the uncovered vertices into three categories: *input-sensitive* vertices with $sbytes(v) \neq \emptyset$, *input-insensitive* vertices with $sbytes(v) = \emptyset$, and vertices with *unknown sensitivity*, on which the sensitivity analysis has not been performed yet. Additionally, we call a vertex *likely input-insensitive* (LII), if it has unknown sensitivity and there is an input-insensitive vertex with the same ABE and calling context in the current ABET.

The input-insensitive vertices often arise in practice. For example, when processing the loop `for (int i = 0; i < 1000; ++i)`, all the ABET vertices with the ABE `i < 1000` will be input-insensitive as the number of iterations does not depend on the input. Moreover, both byteshare and gradient descent analyses are useless on input-insensitive vertices, so we prefer not processing the LII vertices to avoid useless sensitivity computations. However LII vertices cannot be ignored completely as they can be in fact input-sensitive. For this reason, we first try selecting uncovered vertices that are either input-sensitive, or that have unknown sensitivity but are not LII. We sort such vertices lexicographically according to the following criteria and select the best vertex v.

1. Input-sensitive vertices are preferred to vertices with unknown sensitivity as we want to exploit the computed information about sensitive bytes.
2. Vertices with fewer sensitive bytes are preferred, as the analyses are more expensive with more sensitive bytes.
3. Vertices with the number of input bytes closer to the half of the maximal number of input bytes of all ABET vertices are preferred, as it helps to explore loop iterations that are deep enough to be interesting and at the same time to keep the number of input bytes reasonably small.
4. Vertices closer to the root of the execution tree are preferred, as they are easier to process.

If no such vertex exists, we fall back to choosing an LII vertex. We use the distance function to select a promising LII vertex in the following way. We select the uncovered vertex v if it is LII and all identified input-insensitive vertices with the same ABE and context have greater absolute value of the distance function. If there are more such vertices v, we first prefer the ones with the smallest absolute value of the distance function and then according to the criteria similar to the previous ones.

Loop Head Selection Heuristics To fill the worklist H, we detect all loop heads on the path to v. The identified loop heads are grouped to buckets of exponentially increasing size according to the number of bytes read from the input. This ensures that we do not process too many loop heads to make the search impractical, but we still explore loop heads with diverse depths of loop iterations. We then pick from each bucket the vertex that lexicographically minimizes the number of input bytes and the depth and add it to the worklist H.

5.2 Selecting an Open Vertex

If the previous algorithm failed to select a vertex, it means that all uncovered vertices are processed. We try to make progress by selecting a vertex that is covered but still open. The rationale is that by exploring the open vertex, albeit otherwise covered, we hope to extend the ABET with new vertices where the analyses can continue further and some open vertices might become covered.

In particular, we choose an open *input-independent* vertex with a small value of the distance function and identify an earlier loop head on the path to the root as in the previous subsection. We then perform a random ABET traversal from the loop head and select the first open unprocessed vertex for which the search tries to visit to its unvisited child. If this search fails as well, then the analysis cannot make any further progress, returns `null` and the fuzzing loop terminates.

6 Implementation

We implemented the approach in an experimental tool called FIZZER. The tool is implemented in C++, consists of around 11,000 lines of code (in 125 files), and the only external tool it depends on is the CLANG compiler and its libraries. The tool is open-source and available under ZLIB license either as an artifact at Zenodo [13] or at the repository [14].

Given a C program to be analyzed, FIZZER first compiles it into LLVM bitcode using the CLANG compiler. The bitcode is then instrumented using our instrumenter, which first applies a standard LLVM pass to replace all `switch` instructions by sequences of `if-else` statements[2] and then finds and instruments all ABEs and function calls. Observe that we ignore `br` instructions, i.e., we do not care about the actual control flow. After the instrumentation, we link the instrumented LLVM bitcode with our implementations of `nondet_`*type*`()` and `__instr_*()` functions into the final executable program, called `target`, which will be repeatedly executed by the main FIZZER process.

Whenever FIZZER wants to execute the `target` with some input, it spawns a new process with the `target` executable. During the execution of `target`, the instrumented code tracks the current call context, collects data about the executed ABEs, and stores them to the shared memory, which is accessible by the parent FIZZER process. The separation of FIZZER and `target` to independent processes allows handling crashes of the `target`.

[2] We should also replace calls via function pointers by sequences of `if-else` statements. This pass is not implemented yet.

7 Evaluation

Experimental setup. For evaluation of the implemented tool Fizzer, we use all branch-coverage benchmarks from Test-Comp 2023, the 5th Competition on Software Testing [5]. The benchmark set consists of 2933 benchmarks divided into 16 families. For the presentation purposes, *"ReachSafety"* and *"SoftwareSystems"* substrings in the family names are shortened to *"rs"* and *"ss"*, respectively, in the rest of this section. For comparison, we used three best-scoring tools[3] from Test-Comp 2023, namely FuSeBMC [2], VeriFuzz [21], and CoVeriTest [6], in the versions in which they entered Test-Comp 2023. To obtain reproducible results, we asked the organizer of Test-Comp to evaluate Fizzer on the official infrastructure of Test-Comp and compare the obtained results with the official results of Test-Comp 2023. We stress out that this means that the results *were produced by an independent third party* and thus *are independently reproducible*. The resource limits of the competition are 15 minutes of CPU time and 15 GB of RAM. A detailed description of the infrastructure and the setting used for the experimental evaluation we refer to the competition report [5].

Results. The average branch coverage for each tool and each benchmark family is shown in Table 1. The table shows that the approach proposed in this paper and implemented in the tool Fizzer is competitive with FuSeBMC – the winner of Test-Comp 2023 – in most of the benchmark families except *rs-Combinations*, *rs-ECA*, and *rs-Sequentialized*. It is also competitive with the other state-of-the-art tools on all of the benchmark families. Although the table shows that Fizzer is the best on average in benchmark families *rs-ControlFlow* and *ss-SQLite-MemSafety*, we do not consider these particular results significant due to the small size of these families.

Figure 2 provides a comparison of the branch coverage achieved by Fizzer and the other considered tools on individual benchmarks. It can be seen that while on most of the benchmarks, Fizzer provides the same or worse coverage than FuSeBMC, there are some benchmarks where it provides better coverage. It is also comparable with VeriFuzz and provides better branch coverage than CoVeriTest on a large number of benchmarks.

Out of all 2933 evaluated programs, there are 145 programs where Fizzer provides better coverage than any other of the compared tools. For comparison, CoVeriTest provides the best coverage for 129 programs, FuSeBMC for 318, and VeriFuzz for 180. The distribution of these benchmarks to the individual benchmark families can be found in Table 2.

Finally, note that Fizzer participated in Test-Comp 2024 and placed third in the category *Cover-Branches* after FuSeBMC and FuSeBMC-AI.[4]

[3] We do not compare against FuSeBMC IA [1], the runner-up in Test-Comp 2023, as we want to compare only with the best variant of each individual tool, not all their variants.

[4] `https://test-comp.sosy-lab.org/2024/results/results-verified/`

Table 1: *Average branch-coverage* of the tests generated by the individual tools for individual benchmark families and for all benchmarks. The results are in percents. The best result of each benchmark family is printed typeset in bold.

Family	Family size	CoVeriTest	Fizzer	FuSeBMC	VeriFuzz
rs-Arrays	292	71.2	84.6	**86.5**	81.6
rs-BitVectors	61	78.8	77.3	**79.5**	73.8
rs-Combinations	671	34.8	42.0	**50.7**	37.6
rs-ControlFlow	11	4.0	**14.1**	13.7	13.5
rs-ECA	29	18.3	25.1	32.3	**34.9**
rs-Floats	197	46.9	48.2	**50.8**	49.8
rs-Heap	110	68.9	72.5	**72.7**	70.6
rs-Loops	661	79.6	80.3	**82.1**	81.4
rs-ProductLines	263	29.0	28.8	**29.2**	**29.2**
rs-Recursive	51	78.4	84.2	**85.8**	76.0
rs-Sequentialized	91	80.4	66.5	87.8	**88.4**
rs-XCSP	114	**99.8**	88.5	91.7	92.6
ss-BusyBox-MemSafety	62	16.9	32.9	**33.2**	0.0
ss-DeviceDriversLinux64-rs	287	**20.6**	20.5	**20.6**	19.7
ss-SQLite-MemSafety	1	0.0	**3.7**	3.4	3.5
Termination-MainHeap	32	**95.6**	95.3	95.1	90.9
All	2933	54.3	57.3	**61.0**	56.2

8 Related Work

The sensitivity analysis is a form of taint analysis, which is a technique popular in fuzzing [17,8,22,9,4,10,12,23,25,7,19,26]. The most frequent approach to taint analysis is propagating the taint information explicitly from taint sources (e.g., sources of input) through the program instructions [17,8,22,9,4,10,12,23,25]. Most of the approaches propagate taint information dynamically. However, some of them compute it statically [23], with use of control flow information [25], or using concrete and symbolic execution [7]. There are two papers [19,26] that compute the tainted bytes by identifying input bytes that lead to different program executions. This is most similar to our approach. But our approach also tries extreme values of typed inputs and performs more precise one-bit mutations, which are then extended to byte boundaries, while the mentioned papers [19,26] only mutate whole bytes.

Gradient descent is used in fuzzing [8,26,9,16,24] in different forms. For instance, there is a paper [8] that uses forward and backward method of finite differences for computation of the partial derivatives. Additional constraints appearing in the control flow have been also considered [9]. Another approach [16]

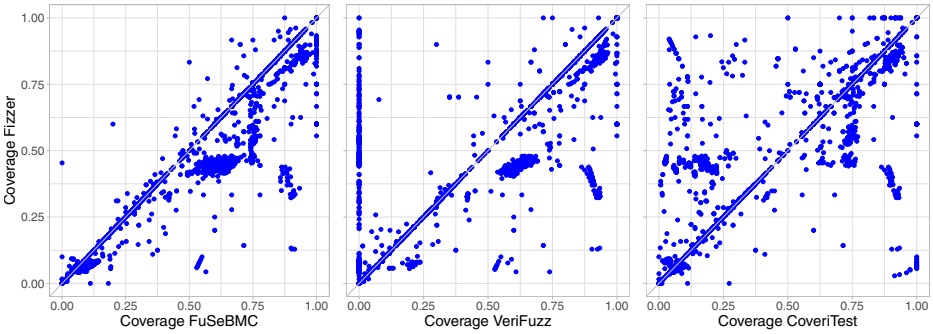

Fig. 2: Scatter plots comparing branch coverage achieved by FIZZER and the other considered tools.

exponentially decreases the learning rate λ as the gradient descent progresses. Our approach differs especially in taking multiple samples along the gradient direction in each descent step. The samples span several orders of magnitude along the line, which can both provide samples in the small region close to the global minimum and help escaping from local minima. We further compute the learning rate λ from the linear approximation of the function. Thanks to multi-sampling, this simplification is sufficient in practice. Lastly, our approach is extended with locking coordinates, which can contribute to escaping from local minima by avoiding extreme directions.

Our approach further uses a unique coverage goal. Other fuzzers monitor actual control flow of the program execution (to measure, e.g., branch coverage), while we ignore it completely. We instead monitor values of all ABEs and aim for their coverage. The byteshare analysis is also a novel approach inspired by genetic algorithms. The random search we apply to select the target vertex is novel among fuzzers, but it was used in the context of concolic execution [20].

The experimental evaluation of the paper compares the proposed approach with the best test-generation tools participating in Test-Comp 2023. All of these combine several analyses. Namely, FUSEBMC [2,3] combines bounded model-checking (BMC), symbolic execution, and two fuzzers (AFL [27] and a selective fuzzer) and FUSEBMC IA [1] extends it further with interval analysis. VERIFUZZ [21] is built on top of AFL and an engine based on Coverage Guided Fuzzing, combined with the bounded model checker CBMC and the PRISM framework. COVERITEST [6] combines several model checkers.

9 Conclusion

We presented a novel approach to gray-box fuzzing, which aims to generate tests that cover both possible values of each atomic Boolean expression. To reach this goal, our approach uses a dynamic computation to identify the bytes that influence the value of a given Boolean expression. Further, it employs two

Table 2: The numbers of benchmarks in individual benchmarks families where a given tool achieved better branch coverage than the other considered tools.

Family	CoVeriTest	Fizzer	FuSeBMC	VeriFuzz
ReachSafety-Arrays	0	12	18	4
ReachSafety-BitVectors	0	3	1	2
ReachSafety-Combinations	96	23	243	139
ReachSafety-ControlFlow	2	1	1	1
ReachSafety-ECA	1	1	6	15
ReachSafety-Floats	0	16	1	0
ReachSafety-Heap	1	8	0	2
ReachSafety-Loops	0	6	6	0
ReachSafety-ProductLines	0	33	0	3
ReachSafety-Recursive	0	1	4	0
ReachSafety-Sequentialized	0	2	16	14
ReachSafety-XCSP	14	0	0	0
SoftwareSystems-BusyBox-MemSafety	4	27	19	0
SoftwareSystems-DeviceDriversLinux64-ReachSafety	9	11	3	0
SoftwareSystems-SQLite-MemSafety	0	1	0	0
Termination-MainHeap	2	0	0	0
All	129	145	318	180

analyses to find the value of these bytes to get the desired value of the Boolean expression. One of these analyses is based on gradient descent.

We implemented the proposed approach in an experimental tool called Fizzer. An independent evaluation shows that, despite being a pure gray-box fuzzer, it is competitive with the state-of-the-art tools competing in Test-Comp 2023.

In future, we plan to add the support for calls via function pointers and gradient descent tailored for floating-point values. We will also investigate an extensible architecture that allows running different external analyses on the vertices of the execution tree. In particular, this would allow running techniques such as symbolic execution on vertices that cannot be covered by gradient descent alone, which could improve the performance of our tool even further.

Acknowledgement

The authors would like to thank Dirk Beyer for running the experiments on the original Test-Comp infrastructure and for his technical assistance.

References

1. Aldughaim, M., Alshmrany, K.M., Gadelha, M.R., de Freitas, R., Cordeiro, L.C.: FuSeBMC_IA: Interval analysis and methods for test case generation (competition contribution). In: Lambers, L., Uchitel, S. (eds.) Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13991, pp. 324–329. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_18, https://doi.org/10.1007/978-3-031-30826-0_18
2. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. In: Loulergue, F., Wotawa, F. (eds.) Tests and Proofs - 15th International Conference, TAP 2021, Held as Part of STAF 2021, Virtual Event, June 21-22, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12740, pp. 85–105. Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6, https://doi.org/10.1007/978-3-030-79379-1_6
3. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC v4: Smart seed generation for hybrid fuzzing. In: Johnsen, E.B., Wimmer, M. (eds.) Fundamental Approaches to Software Engineering. pp. 336–340. Springer International Publishing, Cham (2022)
4. Bekrar, S., Bekrar, C., Groz, R., Mounier, L.: A taint based approach for smart fuzzing. In: Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. p. 818–825. ICST '12, IEEE Computer Society, USA (2012). https://doi.org/10.1109/ICST.2012.182, https://doi.org/10.1109/ICST.2012.182
5. Beyer, D.: Software testing: 5th comparative evaluation: Test-Comp 2023. In: Lambers, L., Uchitel, S. (eds.) Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13991, pp. 309–323. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_17, https://doi.org/10.1007/978-3-031-30826-0_17
6. Beyer, D., Jakobs, M.: Cooperative verifier-based testing with CoVeriTest. Int. J. Softw. Tools Technol. Transf. **23**(3), 313–333 (2021). https://doi.org/10.1007/s10009-020-00587-8, https://doi.org/10.1007/s10009-020-00587-8
7. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive mutational fuzzing. In: 2015 IEEE Symposium on Security and Privacy. pp. 725–741 (2015). https://doi.org/10.1109/SP.2015.50
8. Chen, P., Chen, H.: Angora: Efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 711–725 (2018). https://doi.org/10.1109/SP.2018.00046
9. Chen, P., Liu, J., Chen, H.: Matryoshka: Fuzzing deeply nested branches. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 499–513. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3363225, https://doi.org/10.1145/3319535.3363225
10. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: Proceedings of the 31st International Conference on Software Engineering. p. 474–484. ICSE '09, IEEE Computer Society, USA (2009). https://doi.org/10.1109/ICSE.2009.5070546, https://doi.org/10.1109/ICSE.2009.5070546

11. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. Communications of the ACM **55**(3), 40–44 (2012)
12. Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H.: Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: Proceedings of the 22nd USENIX Conference on Security. p. 49–64. SEC'13, USENIX Association, USA (2013)
13. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Fizzer: Artifact for TACAS 2024 evaluation (Dec 2023). https://doi.org/10.5281/zenodo.10440311
14. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Fizzer: Git repository (2023), https://github.com/staticafi/sbt-fizzer
15. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Gray-box fuzzing via gradient descent and Boolean expression coverage. Tech. rep., Masaryk University, Brno (2024), https://arxiv.org/abs/2401.12643
16. Kim, Y., Yoon, J.: Maxafl: Maximizing code coverage with a gradient-based optimization technique. Electronics **10**(1) (2021). https://doi.org/10.3390/electronics10010011, https://www.mdpi.com/2079-9292/10/1/11
17. Liang, G., Liao, L., Xu, X., Du, J., Li, G., Zhao, H.: Effective fuzzing based on dynamic taint analysis. In: 2013 Ninth International Conference on Computational Intelligence and Security. pp. 615–619 (2013). https://doi.org/10.1109/CIS.2013.135
18. Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J.: Fuzzing: State of the art. IEEE Transactions on Reliability **67**(3), 1199–1218 (2018). https://doi.org/10.1109/TR.2018.2834476
19. Liang, J., Wang, M., Zhou, C., Wu, Z., Jiang, Y., Liu, J., Liu, Z., Sun, J.: PATA: Fuzzing with path aware taint analysis. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 1–17 (2022). https://doi.org/10.1109/SP46214.2022.9833594
20. Liu, D., Ernst, G., Murray, T., Rubinstein, B.I.P.: Legion: Best-first concolic testing. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. p. 54–65. ASE '20, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3324884.3416629, https://doi.org/10.1145/3324884.3416629
21. Metta, R., Yeduru, P., Karmarkar, H., Medicherla, R.K.: VeriFuzz 1.4: Checking for (non-)termination (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 594–599. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_42, https://doi.org/10.1007/978-3-031-30820-8_42
22. Paduraru, C., Melemciuc, M.C., Ghimis, B.: Fuzz testing with dynamic taint analysis based tools for faster code coverage. In: Proceedings of the 14th International Conference on Software Technologies. p. 82–93. ICSOFT 2019, SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT (2019). https://doi.org/10.5220/0007921300820093, https://doi.org/10.5220/0007921300820093
23. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: VUzzer: Application-aware evolutionary fuzzing. In: NDSS. vol. 17, pp. 1–14 (2017)
24. She, D., Pei, K., Epstein, D., Yang, J., Ray, B., Jana, S.: Neuzz: Efficient fuzzing with neural program smoothing. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 803–817. IEEE (2019)

25. Wang, T., Wei, T., Gu, G., Zou, W.: TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE Symposium on Security and Privacy. pp. 497–512 (2010). https://doi.org/10.1109/SP.2010.37
26. You, W., Liu, X., Ma, S., Perry, D., Zhang, X., Liang, B.: SLF: Fuzzing without valid seed inputs. In: Proceedings of the 41st International Conference on Software Engineering. p. 712–723. ICSE '19, IEEE Press (2019). https://doi.org/10.1109/ICSE.2019.00080, https://doi.org/10.1109/ICSE.2019.00080
27. Zalewski, M.: American fuzzy lop (2013), http://lcamtuf.coredump.cx/afl/.