T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax

ZHENYANG XU, University of Waterloo, Canada
YONGQIANG TIAN, The Hong Kong University of Science and Technology, China
MENGXIAO ZHANG, University of Waterloo, Canada
JIARUI ZHANG, University of Waterloo, Canada
PUZHUO LIU, Ant Group, China
YU JIANG, Tsinghua University, China
CHENGNIAN SUN, University of Waterloo, Canada

Program reduction strives to eliminate bug-irrelevant code elements from a bug-triggering program, so that (1) a smaller and more straightforward bug-triggering program can be obtained, (2) and the difference among duplicates (*i.e.*, different programs that trigger the same bug) can be minimized or even eliminated. With such reduction and canonicalization functionality, program reduction facilitates debugging for software, especially language toolchains, such as compilers, interpreters, and debuggers. While many program reduction techniques have been proposed, most of them (especially the language-agnostic ones) overlooked the potential reduction opportunities hidden within tokens. Therefore, their capabilities in terms of reduction and canonicalization are significantly restricted.

To fill this gap, we propose T-Rec, a fine-grained language-agnostic program reduction technique guided by lexical syntax. Instead of treating tokens as atomic and irreducible components, T-Rec introduces a fine-grained reduction process that leverages the lexical syntax of programming languages to effectively explore the reduction opportunities in tokens. Through comprehensive evaluations with versatile benchmark suites, we demonstrate that T-Rec significantly improves the reduction and canonicalization capability of two existing language-agnostic program reducers (*i.e.*, Perses and Vulcan). T-Rec enables Perses and Vulcan to further eliminate 1,294 and 1,315 duplicates in a benchmark suite that contains 3,796 test cases that triggers 46 unique bugs. Additionally, T-Rec can also reduce up to 65.52% and 53.73% bytes in the results of Perses and Vulcan on our multi-lingual benchmark suite, respectively.

Additional Key Words and Phrases: Automated Debugging, Program Reduction, Test Case Minimization

ACM Reference Format:
Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Jiarui Zhang, Puzhuo Liu, Yu Jiang, and Chengnian Sun. 2024. T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax . 1, 1 (August 2024), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program reduction is a useful and widely applied technique in the workflow of testing and debugging language implementations [Donaldson and MacIver 2021; Misherghi and Su 2006; Sun et al. 2018; Zeller and Hildebrandt 2002]. This technique can reduce a bug-triggering program to a minimized version that still manifests the same bug. With such minimized bug-triggering programs, developers can analyze the root cause of the bug without being distracted by the bug-irrelevant code elements

Authors' addresses: Zhenyang Xu, zhenyang.xu@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada, N2L 3G1; Yongqiang Tian, yqtian@ust.hk, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China; Mengxiao Zhang, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada, N2L 3G1, m492zhan@uwaterloo.ca; Jiarui Zhang, j879zhan@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada, N2L 3G1; Puzhuo Liu, liupuzhuo@iie.ac.cn, Ant Group, Hangzhou, Zhejiang, China; Yu Jiang, Tsinghua University, Haidian District, Beijing, China, 100084, jiangyu198964@126.com; Chengnian Sun, cnsun@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada, N2L 3G1.

2024. ACM XXXX-XXXX/2024/8-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

in the original bug-triggering program [Donaldson and MacIver 2021; Misherghi and Su 2006; Sun et al. 2018; Zeller and Hildebrandt 2002]. Moreover, the differences among duplicate bug-triggering programs (*i.e.*, programs that trigger the same bug) are also reduced after reduction. This functionality of program reducers is called canonicalization. Canonicalization is useful and desired, as it significantly benefits bug deduplication and triage [Groce et al. 2017; Regehr 2019; Regehr et al. 2012]. It also benefits the existing approach for solving the compiler fuzzing taming problem [Chen et al. 2013].

Since program reduction is useful and practical for testing and debugging language implementations, most compiler fuzzing work integrates program reduction techniques to process the found bug-triggering programs [Chen et al. 2013; Chong et al. 2015; Le et al. 2014; Livinskii et al. 2020; Sun et al. 2016]. Additionally, many production compilers and interpreters, such as GCC, LLVM, CPython, and JerryScript, have explicitly required bugs be reported with minimized reproducible bug-triggering programs in their bug-tracking systems [CPython 2022; GCC-Wiki 2020; JerryScript 2022; LLVM 2022].

Program reduction has been proven to be an NP-complete problem [Misherghi and Su 2006; Zeller and Hildebrandt 2002] and it is not trivial to find the global minimal program. For effective and efficient reduction, various program reduction techniques have been proposed in recent decades [Kalhauge and Palsberg 2021; Kremer et al. 2021; Misherghi and Su 2006; Niemetz and Biere 2013; Regehr et al. 2012; Sun et al. 2018; Tian et al. 2023; Xu et al. 2023; Zeller and Hildebrandt 2002]. For example, minimizing Delta Debugging (`ddmin`) treats a program as a list of elements. It evenly partitions the list with increasing granularity, and iteratively attempts to delete each partition and the complement from the list until no more elements can be deleted; Hierarchical Delta Debugging (HDD) preprocesses the program into a tree structure, and applies `ddmin` on the tree structure; Perses not only parses the program into a parse tree but also uses the syntax to guide the minimization of the tree. All these techniques are language-agnostic, namely, they are general and applicable to programs in different languages, which is also the focus of this study. [1]

Limitation. Although previous language-agnostic program reduction techniques have been demonstrated to be effective, their performance, especially canonicalization, is still limited. One of the major bottlenecks is related to their coarse reduction granularity. Specifically, in previous techniques such as HDD and Perses, bug-triggering programs are converted into parse trees (*i.e.*, tree-based), and the subsequent reduction transformations (*e.g.*, deletion) regard each leaf node (*i.e.*, each token) as an atomic element. Therefore, each token (such as an identifier and a string literal) can only either be entirely removed or kept as it is. In other words, these algorithms fail to recognize the benefits of shrinking and canonicalizing tokens and consequently do not make attempts to incorporate such optimizations. A recently proposed language-agnostic reducer, DDSET, tackles this problem by converting concrete structures in the program to abstract symbols [Gopinath et al. 2020a]. For example, it can reduce a concrete string token to the abstract string symbol in the corresponding formal grammar, meaning that the content of the string does not affect whether the program triggers the bug or not. However, our evaluation indicates that DDSET is not effective with bug-triggering C programs. One possible explanation is that when the inputs have strict restrictions imposed upon on each token by the semantics or the property (*i.e.*, triggering a certain bug), the abstracting process may become infeasible, and thus the effectiveness of DDSET may significantly deteriorate. C-Reduce [Regehr et al. 2012] is another reducer that attempts to overcome the limitation. It is a widely recognized and powerful reducer commonly employed in compiler

[1] Some of the program reduction approaches are language-specific, *i.e.*, they are optimized with language-specific knowledge to boost the performance in reducing programs in certain languages. [Kalhauge and Palsberg 2021; Kremer et al. 2021; Niemetz and Biere 2013; Regehr et al. 2012]. In this study, we focus on language-agnostic techniques as they do not require domain-specific knowledge, and thus they are more generalized.

testing work and programming language communities [GCC 2020; Le et al. 2014; Li et al. 2023; Livinskii et al. 2020; LLVM 2022; Rigger and Su 2020; Rust 2024; Winterer et al. 2020]. C-Reduce strives to reduce identifiers, strings and numbers to their canonical forms (More details in §3.1.1). However, the approach used by C-Reduce is specifically optimized for reducing C/C++ programs. Although some parts of its approach is language-agnostic, their performance tends to be limited when the language is less similar to C/C++, and thus could benefit from further enhancements.

Practical Benefits of Fine-Grained Reduction. Fine-grained reduction improves program reduction in two aspects. First, it can minimize tokens such as identifiers and literals to canonical tokens (formally defined in §4.1), thus making the bug-triggering program concise and further increasing the degree of canonicalization. Such improvement can benefit bug deduplication, bug triage, and compiler fuzzing taming [Chen et al. 2013]. The other benefit of fine-grained reduction is that it can potentially create reduction opportunities for removing more tokens and lead to smaller and simpler reduction results.

T-Rec. Motivated by the practical benefits above, we propose T-Rec, a *fine-grained* language-agnostic program reduction technique. It enhances the capabilities of reduction and canonicalization of existing language-agnostic program reducers by incorporating a novel, fine-grained reduction phase. Meanwhile, since T-Rec is guided by lexical syntax, it can be readily and effectively applied to a wide range of programming languages. T-Rec reduces tokens in two different ways. First, for each token, T-Rec tries to find the canonical token and replace the original token with the canonical one. Specifically, T-Rec leverages the lexical syntax to generate a list of smaller tokens arranged in ascending shortlex order (*i.e.*, shorter tokens come before longer tokens, and tokens with the same length are ordered lexicographically). Then, it searches for the first token in the list that can replace the original one until an attempt limit is reached. The found token is the canonical token and is used to replace the original token. Such a replacement-based approach is especially effective in scenarios where there is no strict constraint imposed upon the content of the token for the program to trigger the bug. Second, if T-Rec cannot find the canonical token within the attempt limit, T-Rec reduces the token by attempting to remove unnecessary characters in the token with the guidance of lexical syntax and then converts the remaining characters to their canonical forms. More details of this approach are presented in §4. Compared to the replacement-based approach, this method is more effective when some parts of the token are essential for the program to trigger the bug.

We conducted extensive evaluations with two benchmark suites, named `Benchmark-Tamer` and `Benchmark-Multi`. `Benchmark-Tamer` was curated by Chen *et al.* [Chen et al. 2013]. It contains 2,501 test cases that trigger 11 unique crash bugs in GCC 4.3.0, and 1,295 test cases that triggers 35 unique miscompilation bugs also in GCC 4.3.0. We use this benchmark suite for evaluating the performance of T-Rec in facilitating deduplication. `Benchmark-Multi` is a multi-lingual benchmark suite. It contains 20 C, 20 Rust, and 195 SMT-LIBv2 programs in total, and is used for evaluating the effectiveness, efficiency, and generality of T-Rec. We implemented T-Rec on top of Perses [Sun et al. 2018], Vulcan [Xu et al. 2023], and C-Reduce [Regehr et al. 2012], and built three prototypes named T-Rec$_{Perses}$, T-Rec$_{Vulcan}$, and T-Rec$_{C\text{-}Reduce}$. Both Perses and Vulcan are open-source syntax-guided language-agnostic reducers, and C-Reduce is a widely used reducer specifically optimized for reducing C/C++ programs. As for baselines, we use Perses, Vulcan, C-Reduce, DDSET and some variants built on top of these existing reducers (more details in §5).

The evaluation results show that all the reducers with T-Rec integrated exhibit significantly better capability in canonicalization than the original base reducers. Specifically, T-Rec$_{Perses}$, T-Rec$_{Vulcan}$, and T-Rec$_{C\text{-}Reduce}$ eliminate 1,294, 1,315, and 336 more duplicates in `Benchmark-Tamer` than Perses, Vulcan, and C-Reduce. The evaluation results also show that in terms of byte size, on average, the results of T-Rec$_{Perses}$ are 65.52%, 28.34%, and 42.86% smaller than those of Perses,

and the results of T-Rec$_{Vulcan}$ are 53.73%, 19.79%, and 16.24% smaller than those of Vulcan. Such results further demonstrate the superiority of T-Rec in canonicalization. Moreover, in terms of the number of remaining tokens in the reduced program, T-Rec$_{Perses}$ significantly outperforms Perses and C-Reduce-cano$_{Perses}$ (a variant of T-Rec$_{Perses}$ that uses the non-C/C++-specific canonicalization transformations rather than T-Rec in the fine-grained reduction phase), which demonstrates that T-Rec can better create reduction opportunities for removing more tokens than C-Reduce-cano. Finally, the results also show that the overhead brought by T-Rec is within a reasonable range. Concretely, T-Rec$_{Perses}$ takes 24.22%, 31.22%, and 56.46% more time than Perses on C, Rust,and SMT-LIBv2 benchmarks, respectively, and T-Rec$_{Vulcan}$ takes 1.59%, 5.04%, and 5.01% more time than Vulcan on each benchmark.

Contribution. We make the following contributions.

- We propose the first work that uses lexical syntax to efficiently and effectively perform fine-grained language-agnostic program reduction at the token level. Specifically, we propose a two-stage canonicalization approach to reduce each token to its canonical form.
- We demonstrate that the fine-grained reduction process can not only significantly increase the degree of canonicalization for a reducer, but also create more reduction opportunities to further remove tokens, thus enhancing the reduction ability of prior reducers.
- Our extensive experiments with a benchmark suite including programs in C, Rust, and SMT-LIBv2 strongly demonstrate the efficacy and efficiency of T-Rec.
- For reproducibility and replicability, we have released the implementation and data at

  https://github.com/trec-reducer/T-Rec

  T-Rec will be fully open sourced as a part of Perses at https://github.com/uw-pluverse/perses upon acceptance of this manuscript.

2 MOTIVATING EXAMPLE

In the section, we use the benchmarks rust-77002, gcc-60116, and gcc-71626 as motivating examples to illustrate how T-Rec enhances the capabilities of both canonicalization and reduction.

rust-77002. Fig. 1a, 1b, and 1c show the reduced programs output by Perses, C-Reduce-cano$_{Perses}$, and T-Rec$_{Perses}$ on the benchmark rust-77002 respectively. In rust-77002, the bug-triggering program contains an integer token `1_000_000_000_000`. T-Rec successfully reduces it to `0`, while C-Reduce cannot. The reason is that C-Reduce uses regular expression substitutions to canonicalize integers, and the regular expression does not cover integers split with underscores. In contrast, T-Rec can handle any type of integer in any language as it is guided by the lexical syntax. Reducing the original integer token to `0` further narrows down the space of possible root cause of the bug, *i.e.*, it shows the extremely large value of the variable (`1_000_000_000_000`) is not essential to trigger the bug. Additionally, in this case, any program with a different integer is a duplicate (*i.e.*, a different program that triggers the same bug). With T-Rec, the difference among such duplicates can be completely eliminated, which makes deduplication trivial.

gcc-60116. Fig. 1d, 1e, and 1f show the results on the benchmark gcc-60116, which illustrates another two cases where T-Rec performs better than C-Reduce in terms of canonicalization. First, it can be observed that T-Rec reduces the string `"checksum = %X\n"` to `"%X"`, but C-Reduce does not reduce it at all. This is because C-Reduce only attempts to replace the string with an empty string, whereas T-Rec further performs deletion-based canonicalization if such a replacement fails. Additionally, in this benchmark, T-Rec better canonicalizes identifiers than C-Reduce. The original identifier `crc` is reduced to `a` by T-Rec, whereas C-Reduce reduces it to `aa`. Although there is only one-character difference in this case, such improvement can be important in scenarios like bug deduplication. This difference stems from that C-Reduce groups identifier tokens that have the

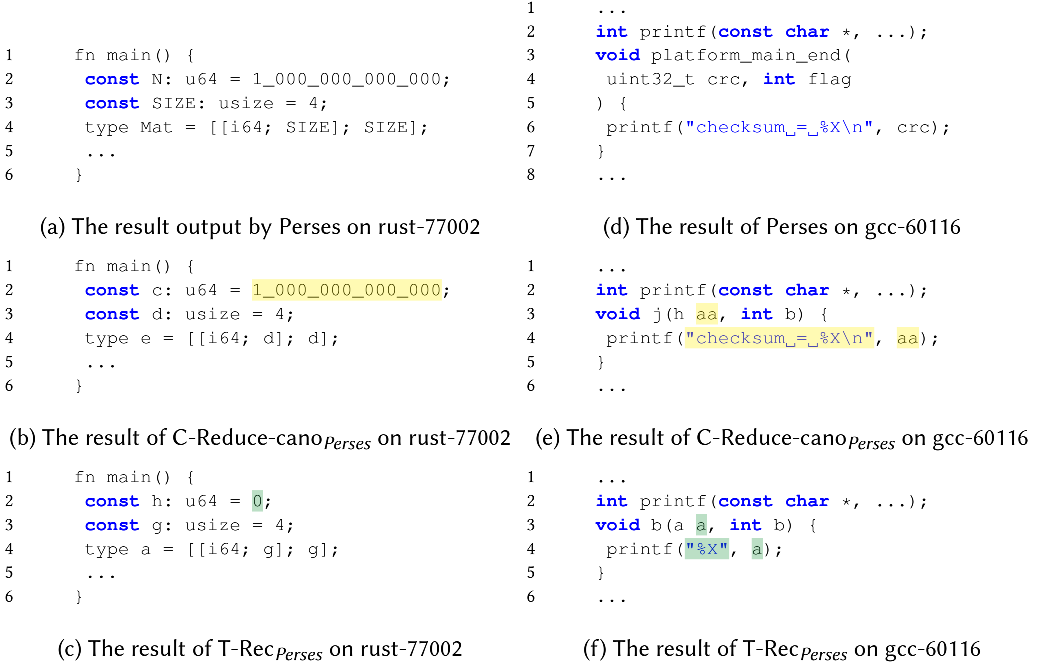

Fig. 1. Code snippets in programs output by Perses, C-Reduce-cano$_{Perses}$, and T-Rec$_{Perses}$ on benchmark rust-77002 (left) and gcc-60116 (right).

same name and renames different groups of identifier tokens with different canonical identifiers. Therefore, the number of unique identifiers does not change after canonicalization. In this case, C-Reduce exhausts all the one-letter identifiers. On the contrary, T-Rec canonicalizes each single token solely, which can potentially decrease the number of unique identifiers.

gcc-71626. Fig. 2 shows the results on the benchmark gcc-71626, which illustrates that T-Rec is effective in removing tokens. Fig. 2a is the reduced program output by Perses, and Fig. 2b is the result of T-Rec$_{Perses}$. Compared to Perses, T-Rec$_{Perses}$ further removed line 2, and 8 (highlighted in red). The reason that Perses cannot remove these two lines is that line 9 in Fig. 2a uses the variable defined by line 8, and line 8 uses the function defined by line 2. Nevertheless, after the fine-grained reduction performed by T-Rec, the identifier in the curly bracket on line 9, becomes the same as the function name on line 7 in Fig. 2a (as shown in Fig. 2b, on line 6 and line 7, the function name and the identifier in the curly bracket are both `b`). This change makes line 2 and 8 in Fig. 2a redundant, and thus they can be removed.

3 PRELIMINARIES

3.1 Program Reduction

A program reduction algorithm can be modeled as a function f that takes two inputs, a program P and a property ψ that P has, and outputs a (possibly) smaller program P' still preserving the property ψ. Mathematically,

$$f(P, \psi) = P' \tag{1}$$

where

$$\psi(P) \wedge \psi(P') \wedge |P'| \leq |P| \tag{2}$$

```
1  typedef long llong;
2  test1char8() {}
3  typedef llong vllong1
4  __attribute__(
5   (__vector_size__(sizeof(llong)))
6  );
7  vllong1 test2llong1() {
8   llong c = test1char8;
9   vllong1 v = {c};
10  return v;
11 }
12 main() {}
```

(a) The result of Perses on gcc-71626

```
1  typedef long a;
2  typedef a c
3  __attribute__(
4   (__vector_size__(sizeof(a)))
5  );
6  c b() {
7   c a = {b};
8   return a;
9  }
10 main() {}
```

(b) The result of T-Rec$_{Perses}$ on gcc-71626

Fig. 2. Results output by Perses and T-Rec$_{Perses}$ on benchmark gcc-71626.

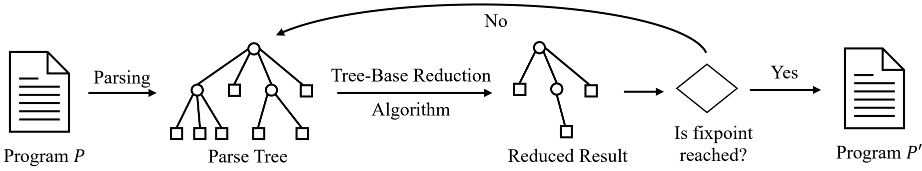

Fig. 3. The overview of tree-based program reduction.

The property ψ is modeled as a predicate, such that for any program P, $\psi(P)$ = true if P preserves ψ and false otherwise. Often, ψ represents whether P triggers a specific bug in a language implementation. The notation $|P|$ represents the size of the program P.

Fixpoint. During the reduction process, it is common that deleting some elements makes some other elements deletable. Therefore, only invoking the algorithm once is usually not sufficient. To achieve the best reduction result within the capacity of the algorithm, a typical way is to repeatedly apply the algorithm to its output until the input program and the output are identical (*i.e.*, $f(P, \psi) = P$) [Hodován et al. 2017; Kiss et al. 2018; Misherghi and Su 2006; Regehr et al. 2012; Sun et al. 2018; Vince 2022]. Such a program that cannot be minimized by the algorithm is called a fixpoint of the program reduction algorithm.

3.1.1 Canonicalization. Another desired functionality that program reduction provides is canonicalization. Ideally, given a property ψ (assuming that ψ is accurate enough to prevent bug slippage [Chen et al. 2013]) and a search space $\mathbb{P}$ that includes all the possible programs, a perfectly canonical program reduction algorithm f has the following property:

$$|\{f(P, \psi) | P \in \mathbb{P} \wedge \psi(P)\}| = 1 \tag{3}$$

This property promises that all the programs in the search space $\mathbb{P}$ that have the same property ψ are reduced to a single, canonical program. In the context of reducing bug-triggering programs, this property means the reducer can reduce all the different programs that trigger the same bug to one canonical bug-triggering program. Such a property is extremely useful in testing and debugging as it significantly facilitates bug deduplication and triage. Although perfectly canonical

program reduction is usually impractical, it is possible and also valuable to increase the degree of canonicalization of a program reducer (*i.e.*, minimizing $|\{f(P, \psi) | P \in \mathbb{P} \wedge \psi(P)\}|$).

Canonicalization in C-Reduce. C-Reduce has a few transformations specifically implemented for canonicalization. These transformations can be classified into three types by their purposes, *i.e.*, renaming identifiers, deleting content of string literals, and reducing integers. All the renaming transformations except one of them are language-specific as they need to be performed on top of the AST. The exception one does not rely on the AST; instead, it refactors the program by renaming all the identifiers with a set of canonical identifiers (*i.e.*, `a`, `b`, `c`, $\cdots$, `aa`, `ab`, `ac`, $\cdots$). The other two types of transformations are also not C/C++-specific. The transformation for deleting the content of strings simply replaces each string token with an empty string, and the transformations for reducing integers are based on regular expression substitution (*e.g.*, matching a hex and replacing with a decimal). For convenience, we refer to the group of the non-C/C++-specific canonicalization transformations mentioned above as **C-Reduce-cano**. However, the generality of C-Reduce-cano is limited even though they are not C/C++-specific. First, the effectiveness of renaming and string-deleting transformation relies on the correctness of the lexing. However, C-Reduce only utilizes a C lexer, which cannot ensure correctness when being applied to different languages. For example, in SMT-LIBv2, identifiers can include dashes (*i.e.*, −), and C-Reduce cannot recognize such identifiers and thus cannot rename them. Second, the regular expression substitution does not work if the target language supports different forms of integer literals. For example, C-Reduce cannot reduce `1_000_000_000_000` to `0` in rust-77002 shown in Fig. 1b. In addition to generality, C-Reduce-cano are also limited in terms of effectiveness. First, C-Reduce-cano cannot effectively canonicalize tokens when there are restrictions imposed upon the tokens. For example, C-Reduce-cano cannot reduce the string at all in gcc-60116 shown in Fig. 1e. Second, C-Reduce-cano can hardly create reduction opportunities for removing more tokens, *e.g.*, they cannot help remove more tokens as T-Rec does in gcc-71626 shown in Fig. 2.

3.1.2 Existing Language-Agnostic Reduction Techniques. The problem of program reduction has been studied for decades. Many reduction techniques have been proposed.

Minimizing Delta Debugging. The minimizing delta debugging algorithm `ddmin` [Zeller and Hildebrandt 2002] is a fundamental algorithm used by a wide range of program reduction tools. Given a bug-triggering input, the intuition of `ddmin` is to consider the input as a list of elements and keep testing whether a partition of the list or the complement of this partition can still trigger the same bug. `ddmin` starts by dividing the list into two partitions and gradually decreases the granularity when none of the partitions or complements can trigger the same bug.

Tree-Based Program Reduction. Tree-based program reduction leverages the tree structure of the program to enhance the reduction performance. Fig. 3 shows the general workflow of tree-based program reduction: the input program is first converted to its tree representation (*e.g.*, its parse tree), and the program reduction algorithm is applied to this tree structure instead of the plain text of the program. For example, HDD runs `ddmin` on each level of the tree representation in a top-down manner.

Syntax-Guided Program Reduction. Besides leveraging the tree structure of programs, syntax-guided program reduction also utilizes formal syntax to guide the reduction. Program reduction is a trial-and-error process. During this process, a large number of programs are generated by deleting part of the original program and tested against the property. The key insight of syntax-guided program reduction is that syntactically invalid programs usually do not have the property (*i.e.*, triggering the same bug as the original program). Therefore, by avoiding generating invalid programs and performing property tests on only syntactically valid ones, much time can be saved.

```
1  Constant
2    : IntegerConstant
3    | FloatingConstant
4    | CharacterConstant ;

6  fragment IntegerConstant
7    : DecConstant IntSuffix?
8    | OctConstant IntSuffix?
9    | HexConstant IntSuffix?
10   | BinaryConstant ;

12 fragment BinaryConstant
13   : '0' [bB] [0-1]+ ;

15 fragment DecConstant
16   : NonzeroDigit Digit* ;

18 fragment OctConstant
19   : '0' OctalDigit* ;

21 fragment HexConstant
22   : HexPrefix HexDigit+ ;

24 fragment HexPrefix
25   : '0' [xX] ;

27 fragment NonzeroDigit
28   : [1-9] ;

29 fragment OctalDigit
30   : [0-7] ;

32 fragment HexDigit
33   : [0-9a-fA-F] ;

35 fragment IntSuffix
36   : USuffix LongSuffix?
37   | USuffix LLSuffix
38   | LongSuffix USuffix?
39   | LLSuffix USuffix? ;

41 fragment USuffix
42   : [uU] ;

44 fragment LongSuffix
45   : [lL] ;

47 fragment LLSuffix
48   : 'll'
49   | 'LL' ;

51 fragment Digit
52   : [0-9] ;
```

Fig. 4. The partial lexical grammar of `Constant` token in C written in ANTLR grammar format

3.2 Lexical Syntax

Lexical syntax determines the way that a sequence of characters should be split into a sequence of tokens. It is described by a list of lexical rules. Each rule defines the form of a specific type of token. Fig. 4 shows a (partial) lexical rule in the syntax of the C programming language written in ANTLR [ANTLR 2017] grammar format, which formally defines the `Constant` token type. The fragment rules (rules that start with the keyword "fragment") are introduced for better readability. This example rule defines that a `Constant` can be an `IntegerConstant`, a `FloatingConstant`, or a `CharacterConstant` (line 1-4). Then, `IntegerConstant` is defined to have four different forms (line 6-10). For example, it can be a `HexConstant` optionally (indicated by `?`) followed by an `IntSuffix` (line 9). `HexConstant`, according to the corresponding fragment rule, is defined to be a `HexPrefix` followed by one or multiple (indicated by `+`) `HexDigit` (line 21-22). A `HexPrefix` can be either `0x` or `0X`, indicated by `'0' [xX]` (line 24-25), while a `HexDigit` can be any character in the set `[0-9a-fA-F]` (line 32-33). Guided by the lexical rule, we can possibly split a token into finer structures. For example, a hexadecimal constant token `0xff00ull`, according to the example lexical rule, can be divided into four components:

Component	Fragment	Description
`0x`	`HexPrefix`	the prefix for hexadecimal numbers
`ff00`	`HexDigit+`	hexadecimal digits
`u`	`USuffix`	the suffix for unsigned numbers
`ll`	`LLSuffix`	the suffix for the integer type `long long`

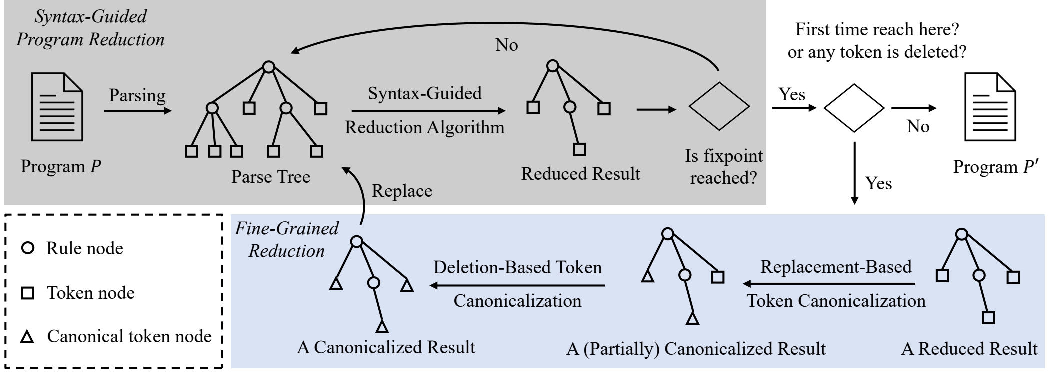

Fig. 5. The overview of T-Rec.

4 METHODOLOGY

Fig. 5 shows the overview of T-Rec. Given a program to reduce, T-Rec first reduces it with a syntax-guided program reduction algorithm. Once the algorithm reaches a fixed point (*i.e.*, it can no longer remove more tokens), the process of fine-grained reduction starts. In this process, T-Rec strives to reduce each token to its canonical form. Specifically, T-Rec first attempts to replace each token with the canonical token, which is generated based on the lexical rule of that certain token type. Such a replacement-based approach is the most efficient way to canonicalize a token. It is also effective in most scenarios, and thus it is performed first. However, some of the tokens may not be able to be directly replaced with the canonical tokens. There might be some constraints imposed upon the tokens to ensure that the program preserves the property (*e.g.*, an integer has to be larger than a certain value). For such tokens, T-Rec further performs a deletion-based approach to canonicalize them. This approach first converts the token to a tree representation that reflects the lexical structure of the token (referred to as a *lex tree*) based on the lexical syntax. Next, it applies lexical syntax-guided reduction to the lex tree to remove unnecessary token fragments. Finally, T-Rec canonicalizes each remaining token fragment by replacing it with the canonical one. This second approach is not as efficient as the first one. Nevertheless, it is more effective when there are strict restrictions imposed upon the token.

Algorithm 1 is the main algorithm of T-Rec. The invocations of `reduceUntilFixpoint` on line 1 and line 13, correspond to the repeating syntax-guided reduction algorithm in Fig. 5. The for loop from line 5 to line 12 corresponds to the fine-grained reduction process, which will be introduced in detail in §4.2 and §4.3. The `while` condition on line 14 ensures that the reduction terminates when the outer fixpoint is reached.

4.1 Definition of Canonical Tokens

Before introducing the two canonicalization approaches, we first define canonical tokens in this subsection.

Definition 4.1 (Token). *A token t consists of a list of characters $[c_1, c_2, ..., c_n]$. It is associated with a type defined by the lexical syntax, denoted as $type(t)$, and its length equals the number of characters in the list* (i.e., *n), denoted as $length(t)$.*

Example. In C programs, `0x00ff` is typically defined as a `Constant` token. Its length is 6 as it contains 6 characters.

Algorithm 1: The Main Algorithm – `reduce` (P, ψ, G)

```
   Input  : P, the current program to be further reduced
   Input  : ψ: ℙ → 𝔹, the property checking function
   Input  : G: the formal syntax of the programming language
   Output : A reduced program that still satisfies ψ
   // syntax-guided program reduction
1  P_min ← reduceUntilFixpoint (P, ψ)
2  do// Token canonicalization loop
3      P_prev ← P_min
4      tokenCount ← P_min.tokenCount()
5      for i ∈ [1, tokenCount] do
6          origToken ← P_min.getNthToken(i)
7          P'_min ← replacementBasedCanonicalize(P_min, origToken, ψ, G)
8          if P'_min ≠ P_min then
9              P_min ← P'_min
10         else
               // deletion-based token canonicalization
11             P_min ← shrinkToken(P_min, origToken, ψ, G)
12             P_min ← canonicalizeTokenFragments(P_min, origToken, ψ, G)
13     P_min ← reduceUntilFixpoint (P_min, ψ)
14 while P_min.size() < P_prev.size()
15 return P_min
```

Definition 4.2 (Shortlex Order). *Given two tokens* $t_1 = [c_1^1, c_2^1, ..., c_m^1]$ *and* $t_2[c_1^2, c_2^2, ..., c_n^2]$, $t_1 <_s t_2$ *if and only if eq.* (4) *or eq.* (5) *holds:*

$$length(t_1) < length(t_2) \tag{4}$$

$$length(t_1) = length(t_2) \land \exists i, \forall j < i \implies c_j^1 = c_j^2 \land c_i^1 < c_i^2 \tag{5}$$

Example. Given three string literal tokens: t_1 is `""`, t_2 is `"a"`, and t_3 is `"b"`, it can be deduced that $t_1 <_s t_2 <_s t_3$ because t_1 contains the smallest number of characters (*i.e.*, two double quotes), and the character `a` in t_2 is in front of the `b` in t_3 in lexicographic order.

Definition 4.3 (Canonical Token). *Given a program P that consists of a list of tokens* $[t_1, t_2, ..., t_n]$ *and a property* ψ *that P has* (i.e., $\psi(P) = \text{true}$)*, a token* t_i *is canonical if and only if:*

$$\forall t_i' <_s t_i \land type(t_i') = type(t_i), P' = P[t_i/t_i'] \implies \psi(P') = \text{false} \tag{6}$$

where $type(t)$ *is the token type of the token t defined by the lexical syntax* (e.g., *identifier), and* $P[t_i/t_i']$ *the program derived by replacing* t_i *with* t_i' *in P.*

Example. Suppose that a program P that contains four tokens, `[print, (, "hello", )]`, can print “hello”, and assume that the property ψ is that the program prints at least one ASCII printable character (character code 32-127). Then, the canonical token of the string token `"hello"` is `"␣"` (*i.e.*, a string that only contains a space), because `"␣"` is the first string token in the shortlex order that preserves the property of the program. Sometimes, following the shortlex order based on ASCII can lead to canonical tokens with poor readability. To fix this limitation, a feasible solution is to customize the order of the alphabet rather than following the conventional order in ASCII.

Algorithm 2: `replacementBasedCanonicalize` (P, origToken, ψ, G)

```
Input  : P, the current best program
Input  : origToken, the token to be replaced
Input  : ψ, the property checking function, which takes as input a program and outputs a boolean
Input  : G: the formal syntax of the programming language
Output : A reduced program that has the property ψ
1  lexicalRule ← getLexicalRule(origToken.tokenType, G)
2  tokenList ← generateInShortlexOrder(lexicalRule)
3  attemptLimit ← getAttemptLimit(origToken.tokenType)
4  for i ∈ [1, min(tokenList.size(), attemptLimit)] do
5      newToken ← tokenList[i]
6      if replaceIfPassTest(P, ψ, origToken, newToken) then
7          break
8  return P
9  Function replaceIfPassTest(P, ψ, origToken, newToken):
       // try to replace tokens with the same text simultaneously
10     P′ ← a new program derived by replacing all the tokens in P that have the same text as origToken
         with newToken
11     if ψ (P′) then
12         update P with P′
13         return true
       // if not work, try to replace a single token
14     P′ ← a new program derived by replacing origToken in P with newToken
15     if ψ (P′) then
16         update P with P′
17         return true
18     return false
```

4.2 Replacement-Based Token Canonicalization

Algorithm 2 describes the way that T-Rec replaces tokens with generated canonical tokens. The algorithm takes four inputs, *i.e.*, the current minimal program, the token to be replaced, the property checking function, and the formal syntax of the corresponding programming language. First, T-Rec retrieves the lexical rule of the token from the given formal syntax based on the token type (line 1). Next, with the obtained lexical rule, it generates a list of tokens in shortlex order (line 2). The generation approach that we use is similar to the input generation method mentioned by a few existing grammar fuzzing studies [Aschermann et al. 2019; Hodován et al. 2018; Srivastava and Payer 2021], but instead of random generation, we make the generation follow the shortlex order. Typically, the result of the generation is determined by a list of choices that decide (1) whether to generate for a part marked with an optional quantifier, (2) how many times to generate a part marked with Kleene star and Kleene plus, (3) and which character(s) within the defined set to generate. The strategy that T-Rec follows is generating as few parts as possible and always prioritizing character(s) that are sequentially positioned in the front in shorlex order. According to the definition of canonical tokens, the first token in tokenList that makes the program pass the property test is the canonical token. Therefore, in the following loop (line 4-7), T-Rec iterates over tokenList until it finds the token that can replace the original token while preserving the property ψ (line 6-7).

Sometimes, the canonical token is not positioned in the front in the shortlex-ordered token list. For example, it is possible that to preserve the property, an integer literal needs to be larger than or equal to 100. In such a case, T-Rec has to try all the integers from 0 to 99 to eventually find that 100 is the canonical token. To ensure the replacement-based token canonicalization algorithm does not waste too much time for such cases, T-Rec terminates the for loop once the attempt limit set for the token type is reached (line 3 and line 4). T-Rec has two different attempt limits for different token types. For identifiers, the attempt limit is set to infinite. Because for identifiers, the aforementioned problem does not exist as the text content of the identifier does not affect the semantics of the program. For tokens other than identifiers, the attempt limit is set to a small value. In our implementation, it is set to 2 in order to cover both 0 and 1 for integer tokens.

Property Checking with New Tokens. The function `replaceIfPassTest` in Algorithm 2 describes the way that T-Rec checks whether a new token can replace the original token(s) without making the program lose the property. As shown in the algorithm, T-Rec performs replacement in two different ways. It first replaces all the tokens in t that have the same text as origToken with newToken to generate a new program (line 10), and it performs the property test on this new program (line 11-13). The insight behind this replacement is that tokens having the same text are often semantically related, and thus replacing them together sometimes better preserves the property of the program. If the new program still has the property (*i.e.*, $\psi(P)$ = true), T-Rec updates the current minimal program P and returns true (line 12-13). Otherwise, T-Rec generates another new program by only replacing the single token origToken in P with newToken (line 14), and T-Rec repeats the previous steps, *i.e.*, testing and possibly updating, with this new program (line 15-17). If neither of these two new programs passes the property, the function returns false, indicating the new token fails the test.

```
1 // The original program
2 // `long_var' on line 6
3 // is the token
4 // to be replaced
5 int main() {
6  int long_var = 1;
7  int b = long_var + 2;
8  return b;
9 }
```

(a) The original program

```
1 // First program to test
2 //
3 // All `long_var's
4 // are replaced
5 int main() {
6     int a = 1;
7     int b = a + 2;
8     return b;
9 }
```

(b) The generated program P_2

```
1 // Second program to test
2 //
3 // Only `long_var' on
4 // line 6 is replaced
5 int main() {
6  int a = 1;
7  int b = long_var + 2;
8  return b;
9 }
```

(c) The generated program P_3

Fig. 6. An example of property checking with a new token

Example. Fig. 6 illustrates the process of a new token being tested by `replaceIfPassTest`. Assuming that the original program is P_1 in Fig. 6a, the original token is `long_var` on line 6, and the new token is `a`. The function `replaceIfPassTest` first generates P_2 as shown in Fig. 6b by replacing all the `long_var` in the original program P_1 with `a`. If P_2 passes the property test ψ, the function updates P with P_2 and returns true to indicate that the test is passed. Otherwise, if P_2 does not pass the property test ψ, the function `replaceIfPassTest` generates P_3 in Fig. 6c by replacing only the `long_var` on line 6, since it is the original token. Next, the function `replaceIfPassTest` tests P_3 against the property test ψ. If the property test is passed, the program P is updated and true is returned; otherwise, the function returns false, indicating the test of this new token fails.

4.3 Deletion-Based Token Canonicalization

Replacement-based token canonicalization is not efficient when the token to be canonicalized is imposed with certain restrictions for preserving the property of the program (*e.g.*, an integer needs to be at least 100 to make the program have the property). Therefore, we propose a deletion-based token canonicalization approach to canonicalize the remaining tokens. The intuition of this approach is to first remove the unnecessary characters or token fragments (*i.e.*, `shrinkToken`) and then canonicalize the remaining token fragments (*i.e.*, `canonicalizeTokenFragments`).

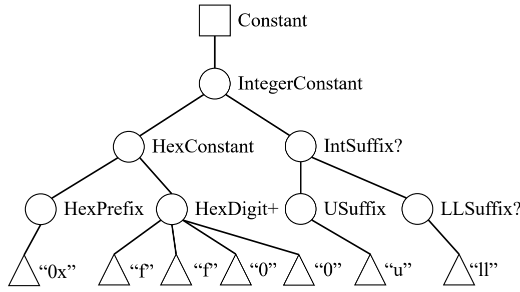

Fig. 7. The lex tree of token `0xff00ull` in C language.

4.3.1 Lex Tree. To facilitate the reduction with lexical syntax, we utilize *lex tree* to represent the internal structures of tokens. *Lex tree* is similar to a parse tree, but instead of representing a program, *lex tree* only represents a single token. Lex trees not only enable T-Rec to perform a tree-based lexical syntax-guided reduction algorithm to effectively and efficiently remove unnecessary parts within a token but also contain the necessary information for canonicalizing the shrunk token.

Fig. 7 shows an example lex tree which is built from a C constant token `0xff00ull`, based on the lexical definition of `Constant`.[2] The root node of the tree represents the corresponding lexical rule. Each intermediate node represents a fragment, and each leaf node is an atomic fragment. Such a tree representation can provide guidance for token shrinking. First, it can be observed that the first two characters `0` and `x` are bound together. They form the atomic fragment `HexPrefix`, and either deleting them or one of them will probably make the token syntactically invalid. Next, `ff00` are four hexadecimal digits. Their parent node is a *Kleene plus node* (indicated by `+`). For every Kleene plus node, its child can be deleted without breaking the syntax as long as at least one child remains. The remaining parts `u` and `ll` are two suffixes of an integer. The former is an `unsigned` suffix and the latter is a `long long` suffix. Their lowest common ancestor is `IntSuffix`, which is an *optional node* (indicated by `?`). For such a node, the sub-tree rooted in it can be deleted without breaking the syntax. Another special type of node that is not included in the example is *Kleene star node* (indicated by `*`). For Kleene star nodes, deleting an arbitrary number of its children does not break the syntax.

In addition to providing guidance to token shrinking, lex trees also contain the information for canonicalization. For example, according to the lex tree, the four characters in `ff00` are all `HexDigit`. According to the formal syntax, each of them can be a character in `[0-9a-fA-F]`. Therefore, we can canonicalize `ff00` by replacing each token fragment with another valid one

[2] A token definition can simply be a regular expression. In such cases, T-Rec recognizes and outlines each necessary, optional, and repetitive fragment in the regular expression and builds lex trees based on the new outlined definition.

that is positioned as close to the beginning as possible in shortlex order, *e.g.*, the resulting four characters can be `1000`.

Algorithm 3: Lexicial Syntax-Guided Reduction – `shrinkToken` (P, origToken, ψ, G)

Input : P, the current best program
Input : origToken, the token to be shrunk
Input : ψ, the property checking function, which takes as input a program and output a boolean
Input : G: the formal syntax of the programming language
Output : A reduced program that has the property ψ

```
1  t_lex ← buildLexTree(origToken, G)
2  queue ← [t_lex.root]
3  while queue is not empty do
4      node ← queue.poll()
5      if node is an optional node then
6          t'_lex ← t_lex.removeNode(node)
7          if replaceIfPassTest(P, ψ, treeToToken(t_lex), treeToToken(t'_lex)) then
8              t_lex ← t'_lex
9          else
10             queue.addAll(node.getChildren())
11     else if node is a Kleene star node or a Kleene plus node then
12         children ← node.getChildren()
13         remaining ← ddmin(children, ψ)
14         update t_lex by removing nodes in children but not in remaining
15         update P with the updated t_lex
16         queue.addAll(remaining)
17     else queue.addAll(node.getChildren())
18 return P
```

4.3.2 *Lexical Syntax-Guided Reduction.* As introduced in §4.3.1, there are three special types of nodes in a lex tree: optional nodes, Kleene star nodes, and Kleene plus nodes. For nodes that belong to these types, deleting them or their children in a certain way can ensure that the resulting token is still syntactically valid. Based on this insight, we propose a lexical syntax-guided reduction algorithm.

Algorithm 3 describes the algorithm in detail. It takes three inputs, the current minimal program P (*i.e.*, the fixpoint result), the token to be shrunk origToken, and the property test function ψ which checks whether the program still has the property. First, T-Rec builds the lex tree for the given token (line 1) and initializes queue with the root node of the lex tree (line 2). Next, T-Rec reduces the lex tree in a `while` loop (line 3-17). Whenever queue is not empty, T-Rec enters the loop and retrieves the first node in the queue (line 4). Depending on the type of the node, T-Rec takes the following actions.

- **Optional Node** If this node is an optional node, T-Rec generates a new lex tree by removing this optional node from the original lex tree (line 6). If the new lex tree t'_{lex} passes the property test, the original lex tree t_{lex} is updated with t'_{lex}, and no nodes are added to the queue (line 7-8); otherwise, if t'_{lex} does not pass the test, T-Rec adds all the children of the optional node to the queue (line 9-10), and t'_{lex} is discarded.

- **Kleene Star Node** or a **Kleene Plus Node** If the node is a Kleene star node or Kleene plus node, T-Rec applies `ddmin` to minimize the number of children of the node. Next, t_{lex} is updated by only keeping the remaining children (line 12-14), and the queue is updated by adding all the remaining children to it (line 16).
- Any nodes that do not belong to these three types are not processed, and T-Rec simply adds their children to the queue (line 17).

4.3.3 Lex Tree-Based Canonicalization. After the lexical syntax-guided reduction finishes, T-Rec canonicalizes the token by replacing each token fragment with another valid one that is positioned as close to the beginning as possible in shortlex order as shown in Algorithm 4 (line 3-10). The canonicalization process is similar to replacement-based token canonicalization. Replacement-based token canonicalization canonicalizes the entire program by replacing each token with a canonical one, while lex tree-based canonicalization canonicalizes a single token by replacing each token fragment with a canonical one. Note that there is also an attempt limit in the algorithm (line 6). In our implementation, we set it to 2, which is the same value in replacement-based token canonicalization for tokens other than identifiers.

Algorithm 4: `canonicalizeTokenFragments` (P, origToken, ψ, G)

Input : P, the current best program
Input : origToken, the token to be canonicalized
Input : ψ, the property checking function, which takes as input a program and output a boolean
Input : G: the formal syntax of the programming language
Output : A canonicalizaed program that has the property ψ

```
1  t_lex ← buildLexTree(origToken, G)
2  leaves ← t_lex.getLeaves()
3  for i ∈ [1, leaves.size()] do
4      fragment ← leaves(i)
5      fragmentList ← generateFragmentsInShortlexOrder(t_lex, fragment, G)
6      for j in range(min(fragmentList.size(), attemptLimit)) do
7          newFragment ← fragmentList[j]
8          t'_lex ← a new lex tree derived by replacing fragment with newFragment
9          if replaceIfPassTest(P, ψ, treeToToken(t_lex), treeToToken(t'_lex)) then
10             break
11 return P
```

Example. To illustrate the process of the deletion-based token canonicalization step by step, we use `0xff00ull` as an illustrating example and show how this token is canonicalized. The process is shown in Fig. 8, using the lex tree of `0xff00ull` shown in Fig. 7. The lexical syntax-guided reduction algorithm visits nodes in the breadth-first order, and thus `IntSuffix?` is the first special node being visited. Since it is an optional node, the algorithm removes the entire node, which creates a new token `0xff00` (step ①), and invokes `replaceIfPassTest` to test this new token. Assuming the new token fails the test, the algorithm discards the new token, and continues to visit subsequent nodes. The next special node is `HexDigit+`. For this Kleene plus node, the algorithm invokes `ddmin` to reduce the four children (step ② - ④). Assuming that eventually `ddmin` removes the last `HexDigit` (*i.e.*, 0) (to ensure the result is a fixpoint, more steps are performed by `ddmin` after step ④, we assume all these steps fail the property test and omit them for brevity), the token is then updated with the smaller one, and the program

Original Token	0xff00ull		New Token		ψ		Updated Token
① remove `IntSuffix?`:	0xff00~~ull~~	→	0xff00	→	False	→	0xff00ull
② reduce `HexDigit+`:	0xff~~00~~ull	→	0xffull	→	False	→	0xff00ull
③ reduce `HexDigit+`:	0x~~ff~~00ull	→	0x00ull	→	False	→	0xff00ull
④ reduce `HexDigit+`:	0xff0~~0~~ull	→	0xff0ull	→	True	→	0xff0ull
⑤ remove `LLSuffix?`:	0xff0u~~ll~~	→	0xff0u	→	True	→	0xff0u
⑥ replace 1st `HexDigit`:	0xff0u	→	0x0f0u	→	False	→	0xff0u
⑦ replace 1st `HexDigit`:	0xff0u	→	0x1f0u	→	True	→	0x1f0u
⑧ replace 2nd `HexDigit`:	0x1f0u	→	0x100u	→	True	→	0x100u

Fig. 8. An example showing the process of the deletion-based token canonicalization.

is updated accordingly. The last special node is `LLSuffix?`, which is also an optional node. Similarly, the algorithm generates a new token by removing the node. If this new token passes the test, the algorithm updates the lex tree and the program again. Eventually, the lexical syntax-guided reduction algorithm shrinks the token from `0xff00ull` to `0xff0u`. Next, T-Rec performs the lex tree-based canonicalization algorithm. Since the first token fragment `0x` does not have any other valid alternatives, T-Rec just skips it and moves to the next token fragment, *i.e.*, the first `f`. From the formal syntax, this fragment is defined by `[0-9a-fA-F]`. The first two fragments in this set in shortlex order are `0` and `1`. Therefore, T-Rec first attempts to replace the first `f` with `0` (step ⑥). If such a replacement fails, T-Rec attempts to replace it with `1` (step ⑦). Suppose such a replacement passes the property test, T-Rec then updates the token and the program and moves to the next fragment and repeats the same process (step ⑧).

5 EVALUATION

To demonstrate the effectiveness and efficiency of T-Rec, we implemented prototypes of T-Rec on top of Perses, Vulcan, and C-Reduce named T-Rec$_{Perses}$, T-Rec$_{Vulcan}$, T-Rec$_{C\text{-}Reduce}$[3] respectively and conducted extensive evaluations to investigate the following research questions:
RQ1: What is the effectiveness of T-Rec in facilitating dedupliation?
RQ2: What is the effectiveness of T-Rec in reducing byte size?
RQ3: What is the effectiveness of T-Rec in removing tokens?
RQ4: How efficient is T-Rec?

Benchmark Suites. Two benchmarks suites are used for evaluating T-Rec. The first benchmark suite is collected and used by Chen *et al.* [Chen et al. 2013] (referred to as `Benchmark-Tamer`). We use this benchmark for evaluating the performance of T-Rec in facilitating deduplication. The other benchmark suite used for evaluating T-Rec is named `Benchmark-Multi`. This benchmark suite contains bug-triggering programs in three different programming languages, *i.e.*, C, Rust, and SMT-LIBv2. It is included to further assess the effectiveness and efficiency of T-Rec. Meanwhile, this multi-lingual benchmark suite also allows us to evaluate the generality of T-Rec to some extent.

- `Benchmark-Tamer` includes 2,501 test cases that trigger 11 unique crash bugs and 1,295 test cases (1,298 in total, but we cannot reproduce the bug with 3 test cases) that trigger 35 unique miscompilation bugs in GCC 4.3.0. All the test cases are generated by CSmith [Yang

[3]T-Rec$_{C\text{-}Reduce}$ is implemented by replacing C-Reduce-cano (*i.e.*, the non-C/C++-specific canonicalization transformations) in C-Reduce with T-Rec

et al. 2011] version 2.1.0 under its default setting. This benchmark suite also contains JavaScript bug-triggering test cases. We did not use them since we did not have access to the corresponding property test scripts.

- `Benchmark-Multi` consists of 20 C programs, 20 Rust programs, and 195 SMT-LIBv2 programs. The C programs are also used for evaluation in previous studies [Gharachorlu and Sumner 2023; Sun et al. 2018; Wang et al. 2021; Xu et al. 2023; Zhang et al. 2024, 2023], which is generated by fuzzing techniques including CSmith and EMI [Le et al. 2014]. The Rust benchmarks are all real-world bugs collected by the authors, and most of them are also used for evaluation in previous research studies [Xu et al. 2023; Zhang et al. 2024, 2023]. Each C and Rust benchmark consists of a bug-triggering program that triggers an unique crash or miscompilation bug and a shell script that checks whether a program triggers the bug in certain affected version(s) of compilers. The SMT-LIBv2 programs are collected from the benchmark suite for evaluating ddSMT [Kremer et al. 2021; Niemetz and Biere 2013], which contains 241 SMT-LIBv2 programs in total. We used 195 of them in our evaluation since the remaining 46 SMT-LIBv2 programs cannot be parsed or pass the property check.

Baselines. The baselines to which we compared T-Rec include Perses, Vulcan, C-Reduce, DDSET, and a few variants built based on these reducers.

- **Perses** is a language-agnostic syntax-guided program reducer [Sun et al. 2018]. It excels at efficiency, and reducers including Vulcan and DDSET are built on top of Perses.
- **Vulcan** is the state-of-the-art language-agnostic program reducer [Xu et al. 2023]. It integrates three auxiliary reducers on top of Perses to trade off execution time for smaller outcomes.
- **DDSET** is another language-agnostic reducer built on top of Perses [Gopinath et al. 2020a]. It strives to abstract the bug-triggering input after reduction, which can be considered as a different approach of canonicalization. We use the implementation from the DDSET GitHub repository [Gopinath et al. 2020b] and follow the instructions provided in the repository.
- **C-Reduce** is a commonly used state-of-the-art program reducer that is specifically optimized for reducing C/C++ programs. For C-Reduce, in addition to its default setting (referred to as C-Reduce), we also include C-Reduce with its `sllooww` flag enabled (C-Reduce under this setting is referred to as **C-Reduce-slow**) as a baseline. When the flag `sllooww` is enabled, C-Reduce performs more exhaustive reduction transformations and thus having enhanced capability of both reduction and canonicalization at the cost of execution time.
- **C-Reduce-cano$_{Perses}$** and **C-Reduce-cano$_{Vulcan}$** are two variants of T-Rec$_{Perses}$ and T-Rec$_{Vulcan}$. They are built by replacing the find-grained reduction components (blue part in Fig. 5) with C-Reduce-cano.

The name convention we followed for naming the prototypes and variants (reducer with a subscript in its name) is that the base name refers to the canonicalization approach, and the subscript refers to the main reduction algorithm.

Metrics. In the evaluation, the following four metrics are used to measure the performance of program reduction:

- **Number of Eliminated Programs**: the total number of eliminated programs. Given a list of programs, after reduction, some programs may become textually equivalent. For each group of textually equivalent programs, only one of them is kept and others are eliminated. A higher value of this metric indicates a better capability of the corresponding reducer in canonicalization.
- **Byte Size**: the size in byte of the reduced program. This metric can reflect the capability of both reduction and canonicalization to some extent. The smaller the byte size, the better the performance in reduction and canonicalization.

- **Token Count**: the number of tokens in the reduced program. This metric mainly measures the capability of reduction. The smaller the number of tokens, the better the performance in reduction.
- **Time**: the execution time (in seconds) of the reduction.

We measure both the byte size and token count of the results to provide a comprehensive evaluation of the effectiveness. The number of tokens reflects the reduction capability of a reducer. Fewer tokens in the output program indicate better reduction capability. While the byte size also reflects the reduction capability, it, meanwhile, reveals the capability in canonicalization of a reducer to some extent. Having fewer redundant bytes in the reduced program often leads to fewer differences among reduced duplicates. To observe the reduction performance gap between T-Rec and the baselines in a straightforward way, we also include **percentage change in size** (for both byte size and token count) in the evaluation results, which is calculated by $(\text{size} - \text{size}_{\text{baseline}})/\text{size}_{\text{baseline}}$.

Different tools may output reduced programs in different formats, and the byte size of the programs can be noticeably affected by their format. To eliminate the influence of format on size comparison, we do not count any whitespace in the program.

Experiment Setup. The experiments are run on an Ubuntu 20.04 server with AMD Ryzen 9 7950X 16-Core CPU and 128 GB RAM. All the experiments are run with a single thread.

Table 1. The reduction results produced by different reducer on the 2,501 crash test cases in `Benchmark-Tamer`. The column **Eliminated (#)** shows the total number of programs eliminated by each reducer.

Reducer	Eliminated (#)	Average Time (s)	Average Size (# of Tokens)
Perses	3	83.05	59.87
C-Reduce-cano$_{Perses}$	741	84.52	59.87
T-Rec$_{Perses}$	1,297	89.76	55.73
Vulcan	13	146.97	42.27
C-Reduce-cano$_{Vulcan}$	808	148.05	42.27
T-Rec$_{Vulcan}$	1,327	157.71	40.80
C-Reduce	1,143	287.09	35.29
T-Rec$_{C\text{-}Reduce}$	1,466	375.48	33.90
C-Reduce-slow	1,959	768.67	31.73
T-Rec$_{C\text{-}Reduce\text{-}slow}$	2,053	778.68	30.34
DDSET	3	667.51	N/A

5.1 RQ1: Effectiveness in Deduplication

A highly canonicalizing reducer tends to reduce different test cases that triggers the same bug to similar or even identical test cases. Therefore, one meaningful perspective to evaluate the capability of a reducer in canonicalization is to evaluate whether it can reduce duplicates (*i.e.*, test cases that trigger the same bug) to a textually identical test case. In this experiment, we compare the performance of different program reducers in deduplication using `Benchmark-Tamer`. Specifically, we utilize each reducer to reduce all the test cases in the benchmark suite. Among the reduced test cases, if there are textually equivalent test cases, only one of them is kept and the others are eliminated. We measure the number of eliminated test cases, the average time to reduce, and the

Table 2. The reduction results produced by different reducer on the 1,295 miscompilation test cases in `Benchmark-Tamer`. The Eliminated (#) column shows the total number of programs that are eliminated because they are textually equivalent to other programs.

Reducer	Eliminated (#)	Average Time (s)	Average Size (# of Tokens)
Perses	0	394.36	285.03
C-Reduce-cano$_{Perses}$	0	430.09	284.42
T-Rec$_{Perses}$	0	511.41	228.59
Vulcan	0	2,134.71	197.72
C-Reduce-cano$_{Vulcan}$	0	2,146.40	197.17
T-Rec$_{Vulcan}$	1	2,223.95	188.88
C-Reduce	36	1,247.19	79.26
T-Rec$_{C\text{-}Reduce}$	49	1,304.69	73.43
C-Reduce-slow	129	3,468.54	74.79
T-Rec$_{C\text{-}Reduce\text{-}slow}$	163	3,511.17	69.14
DDSET	0	5,203.80	N/A

average size of the reduced test cases for each reducer. Table 1 and Table 2 show the results of the experiment.

***Perses* v.s. *C-Reduce-cano*$_{Perses}$ v.s. *T-Rec*$_{Perses}$.** T-Rec$_{Perses}$ is a prototype of T-Rec implemented on top of Perses, and C-Reduce-cano$_{Perses}$ integrates C-Reduce-cano instead of T-Rec on top of Perses to serve as the fine-grained reduction technique. As shown in Table 1, C-Reduce-cano$_{Perses}$ and T-Rec$_{Perses}$ eliminate 741 and 1,297 textually equivalent programs after reducing the 2,501 crash test cases, whereas Perses only eliminates 3 programs. In terms of the 1,295 miscompilation test cases, none of these three reducers successfully produces textually equivalent programs and thus no program is eliminated. Such a result indicates that both C-Reduce-cano and T-Rec can effectively canonicalize programs to some extent, and T-Rec is even more effective. Additionally, it should be noted that the results of T-Rec$_{Perses}$ contain fewer tokens than Perses. Specifically, on average, the results of T-Rec$_{Perses}$ have 5.01% and 18.52% fewer tokens than those of Perses on crash and miscompilation test cases, respectively. Timewise, C-Reduce-cano$_{Perses}$ and T-Rec$_{Perses}$ spend 2.91% and 14.79% more time than Perses on the crash test cases and 10.18% and 29.62% on the miscompilation test cases.

***Vulcan* v.s. *C-Reduce-cano*$_{Vulcan}$ v.s. *T-Rec*$_{Vulcan}$.** Similar to C-Reduce-cano$_{Perses}$ and T-Rec$_{Perses}$, C-Reduce-cano$_{Vulcan}$ and T-Rec$_{Vulcan}$ integrate C-Reduce-cano and T-Rec on top of Vulcan, respectively. The comparison among these three reducers further supports the conclusion that both C-Reduce-cano and T-Rec can effectively canonicalize programs to some extent, and T-Rec is more effective than C-Reduce-cano. Specifically, T-Rec$_{Vulcan}$ can eliminate 519 (= 1327 − 808) more crash test cases than C-Reduce-cano$_{Vulcan}$, and among the three reducers, T-Rec$_{Vulcan}$ is the only one that can eliminate a miscompilation test case. Timewise, T-Rec$_{Vulcan}$ takes 8.49% and 5.77% more time than Vulcan on crash and miscompilation test cases, respectively.

***C-Reduce and C-Reduce-slow* v.s. *T-Rec*$_{C\text{-}Reduce}$ *and T-Rec*$_{C\text{-}Reduce\text{-}slow}$.** To further evaluate the capability of T-Rec in deduplication, we implement T-Rec$_{C\text{-}Reduce}$ by replacing C-Reduce-cano with T-Rec, and compare this prototype to C-Reduce with the default setting and with `sllooww` flag enabled. As shown in Table 1 and Table 2, the T-Rec technique helps C-Reduce and C-Reduce-slow further eliminate 323 and 94 crash test cases as well as 13 and 34 miscompilation test cases.

Moreover, on average, the results of T-Rec$_{\textit{C-Reduce}}$ contain 2.84% and 6.04% fewer tokens than those of C-Reduce on crash test cases and miscompilation test cases, respectively. Correspondingly, the percentage decrease in token size of T-Rec$_{\textit{C-Reduce-slow}}$ compared to C-Reduce-slow is 2.84% and 4.20%, respectively. In terms of execution time, T-Rec$_{\textit{C-Reduce}}$ spends 37.48% and 5.12% more time than C-Reduce, and T-Rec$_{\textit{C-Reduce-slow}}$ spends 1.40% and 1.36% more time than C-Reduce-slow on reducing crash test cases and miscompilation test cases, respectively.

Comparing with DDSET. As shown in Table 1, DDSET cannot eliminate more crash test cases than Perses. Moreover, its average execution time is 8.04× that of Perses. For DDSET, its results contains abstract symbols (*i.e.*, terminals and nonterminals) and cannot be directly count by our tool. Considering that token count is not the focus of this experiment, and it is not clear what is the proper way to count abstract symbols, we did not include the token count results for DDSET. In terms of reducing miscompilation test cases, as shown in Table 2, DDSET cannot eliminate any test case, and its average execution time is the highest among all the reducers. It should be noted that DDSET exceeds the time limit (set to 7,200 seconds) for 897 test cases, and the results shown in the table do not include these timeout results. To understand the reason why DDSET is not effective in this task, we investigated a few results produced by DDSET and found that the structures that DDSET can abstract is very limited. A possible explanation is that these C programs are too complex for DDSET to abstract. In the DDSET paper, the benchmark suite used for evaluation includes four input languages, *i.e.*, JavaScript, Clojure, Lua, and UNIX command line utilities including grep and find, and the bug-triggering inputs after the reduction by Perses (with which the abstracting process starts) are relatively small (46.82 characters on average and 185 characters at most). In contrast, the crash test cases reduced by Perses contains 244.94 bytes on average and 3,080 bytes at most. Moreover, the restrictions imposed upon the elements of a C test case by the strict semantic requirements and the property (*i.e.*, triggering the bug) may make the abstracting process infeasible.

> **RQ1**: T-Rec can enable previous reducers to eliminates more duplicates by reducing them to textually equivalent test cases. Specifically, T-Rec$_{\textit{Perses}}$, T-Rec$_{\textit{Vulcan}}$, T-Rec$_{\textit{C-Reduce}}$, and T-Rec$_{\textit{C-Reduce-slow}}$ eliminate 1,294, 1,315, 336, and 128 more duplicates (including eliminated crash test cases and miscompilation test cases) than Perses, Vulcan, C-Reduce, and C-Reduce-slow, respectively.

5.2 RQ2: Effectiveness in Reducing Byte Size

The first research question has already demonstrated that T-Rec is effective in dedupliation, which indicates that T-Rec is effective in canonicalization to some extent. In this research question, we evaluate the canonicalization capability of T-Rec from another angle. Since the canonical token in this work is defined with shorlex order. The byte size of the reduced program reflects the degree of canonicalization to some extent. A reducer with higher degree of canonicalization is expected to produce results with smaller byte size. Therefore, in this experiment, we measure the byte sizes of the results produced by each tool on `Benchmark-Multi`. The results are shown in Table 3 and Fig. 9.

T-Rec$_{\textit{Perses}}$* v.s. *C-Reduce-cano$_{\textit{Perses}}$* v.s. *Perses. In this experiment, T-Rec$_{\textit{Perses}}$ significantly outperforms both Perses and C-Reduce-cano$_{\textit{Perses}}$. In 199 out of 235 benchmarks, the result of T-Rec$_{\textit{Perses}}$ is smaller than both that of Perses and C-Reduce-cano$_{\textit{Perses}}$, and in the remaining benchmarks, the results of T-Rec$_{\textit{Perses}}$ and C-Reduce-cano$_{\textit{Perses}}$ have the same sizes. On average, the result of T-Rec$_{\textit{Perses}}$ is 65.52% (and 27.04%), 28.34% (and 8.23%), and 42.86% (and 35.39%) smaller than that of Perses (and C-Reduce-cano$_{\textit{Perses}}$) on C, Rust, and SMT-LIBv2 benchmarks, respectively. It should

Table 3. The byte sizes of the results produced by different reducers on C and Rust benchmarks.

Subjects	Perses	C-Reduce-cano$_{Perses}$	T-Rec$_{Perses}$	Change *w.r.t.* Perses	Chance *w.r.t.* C-Reduce-cano$_{Perses}$	Vulcan	C-Reduce-cano$_{Vulcan}$	T-Rec$_{Vulcan}$	Change *w.r.t.* Vulcan	Change *w.r.t.* C-Reduce-cano$_{Vulcan}$
clang-22382	524	285	231	-55.92%	-18.95%	344	189	189	-45.06%	0.00%
clang-22704	246	124	117	-52.44%	-5.65%	193	94	94	-51.30%	0.00%
clang-23309	1,899	716	153	-91.94%	-78.63%	1,237	465	431	-65.16%	-7.31%
clang-23353	331	169	152	-54.08%	-10.06%	308	150	150	-51.30%	0.00%
clang-25900	926	412	328	-64.58%	-20.39%	353	149	149	-57.79%	0.00%
clang-26760	487	193	115	-76.39%	-40.41%	183	103	104	-43.17%	0.97%
clang-27137	684	308	80	-88.30%	-74.03%	354	141	126	-64.41%	-10.64%
clang-27747	415	193	128	-69.16%	-33.68%	301	116	116	-61.46%	0.00%
clang-31259	1,341	617	424	-68.38%	-31.28%	843	390	359	-57.41%	-7.95%
gcc-59903	906	464	446	-50.77%	-3.88%	518	277	276	-46.72%	-0.36%
gcc-60116	1,901	764	455	-76.07%	-40.45%	739	387	344	-53.45%	-11.11%
gcc-61383	980	416	308	-68.57%	-25.96%	649	264	265	-59.17%	0.38%
gcc-61917	557	250	199	-64.27%	-20.40%	319	144	145	-54.55%	0.69%
gcc-64990	733	436	314	-57.16%	-27.98%	553	271	270	-51.18%	-0.37%
gcc-65383	516	223	186	-63.95%	-16.59%	241	115	119	-50.62%	3.48%
gcc-66186	1,128	473	396	-64.89%	-16.28%	668	288	290	-56.59%	0.69%
gcc-66375	1,596	671	491	-69.24%	-26.83%	824	335	318	-61.41%	-5.07%
gcc-70127	1,000	503	339	-66.10%	-32.60%	768	334	317	-58.72%	-5.09%
gcc-70586	633	238	221	-65.09%	-7.14%	326	157	158	-51.53%	0.64%
gcc-71626	167	105	95	-43.11%	-9.52%	137	91	91	-33.58%	0.00%
Average	848.5	378	258.9	-65.52%	-27.04%	492.9	223	215.55	-53.73%	-2.05%
rust-111502	429	299	292	-31.93%	-2.34%	395	284	282	-28.61%	-0.70%
rust-112061	1,002	951	727	-27.45%	-23.55%	963	912	757	-21.39%	-17.00%
rust-112213	1,509	1,335	1,068	-29.22%	-20.00%	1,323	1,171	1,062	-19.73%	-9.31%
rust-112526	996	984	804	-19.28%	-18.29%	900	888	726	-19.33%	-18.24%
rust-44800	1,240	850	759	-38.79%	-10.71%	795	569	524	-34.09%	-7.91%
rust-66851	1,767	1,191	1,126	-36.28%	-5.46%	1,791	1,294	1,170	-34.67%	-9.58%
rust-69039	377	274	274	-27.32%	0.00%	295	267	259	-12.20%	-3.00%
rust-77002	452	368	328	-27.43%	-10.87%	468	380	360	-23.08%	-5.26%
rust-77320	84	73	73	-13.10%	0.00%	86	78	74	-13.95%	-5.13%
rust-77323	49	49	49	0.00%	0.00%	49	49	49	0.00%	0.00%
rust-77910	62	53	43	-30.65%	-18.87%	43	39	37	-13.95%	-5.13%
rust-77919	176	135	135	-23.30%	0.00%	172	141	132	-23.26%	-6.38%
rust-78005	188	157	157	-16.49%	0.00%	218	209	157	-27.98%	-24.88%
rust-78325	95	63	63	-33.68%	0.00%	92	60	60	-34.78%	0.00%
rust-78651	53	34	34	-35.85%	0.00%	24	24	24	0.00%	0.00%
rust-78652	230	139	134	-41.74%	-3.60%	141	127	122	-13.48%	-3.94%
rust-78655	50	41	41	-18.00%	0.00%	49	43	41	-16.33%	-4.65%
rust-78720	174	100	100	-42.53%	0.00%	107	86	86	-19.63%	0.00%
rust-91725	338	255	156	-53.85%	-38.82%	208	161	155	-25.48%	-3.73%
rust-99830	685	623	548	-20.00%	-12.04%	632	580	544	-13.92%	-6.21%
Average	497.8	398.7	345.55	-28.34%	-8.23%	437.55	368.1	331.05	-19.79%	-6.55%

be noted that part of the decrease in byte size achieved by T-Rec$_{Perses}$ is attributed to the fact that T-Rec$_{Perses}$ removes more tokens than Perses.

***T-Rec$_{Vulcan}$* v.s. *Vulcan*.** Compared to Perses, the reduction algorithm of Vulcan is more exhaustive, and T-Rec$_{Vulcan}$ fails to remove more tokens than Vulcan (as shown in §5.3). Nevertheless, T-Rec$_{Vulcan}$ still manages to produce 53.73%, 19.79%, and 16.24% smaller outcomes than Vulcan on C, Rust, and SMT-LIBv2 benchmarks, respectively. Such a result indicates that T-Rec is effective in increasing the degree of canonicalization.

***T-Rec$_{Vulcan}$* v.s. *C-Reduce-cano$_{Vulcan}$*.** Although C-Reduce-cano$_{Vulcan}$ is also effective in terms of increasing the degree of canonicalization, T-Rec$_{Vulcan}$ significantly outperforms C-Reduce-cano$_{Vulcan}$ on Rust and SMT-LIBv2 benchmarks. The p-values yielded by the Wilcoxon signed-rank test [Wilcoxon 1992] are 4.34×10^{-4} and 1.36×10^{-4}, respectively. This result demonstrates that T-Rec has better generality than C-Reduce-cano. On C benchmarks, T-Rec$_{Vulcan}$ performs slightly better than C-Reduce-cano$_{Vulcan}$, but the improvement is not significant. The p-value is 0.14. We also noticed that in 6 out of 20 C benchmarks and 7 out of 195 SMT-LIBv2 benchmarks, the results of C-Reduce-cano$_{Vulcan}$ are smaller than those of T-Rec$_{Vulcan}$, and we manually investigated each of them. For the 6 C benchmarks, the differences are due to that the passes of C-Reduce sometimes can reduce an integer to a smaller one than T-Rec (*e.g.*, it can convert a hex to a decimal, thus removing `0x`). For the 7 SMT-LIBv2 benchmarks, the differences are all due to that T-Rec considers

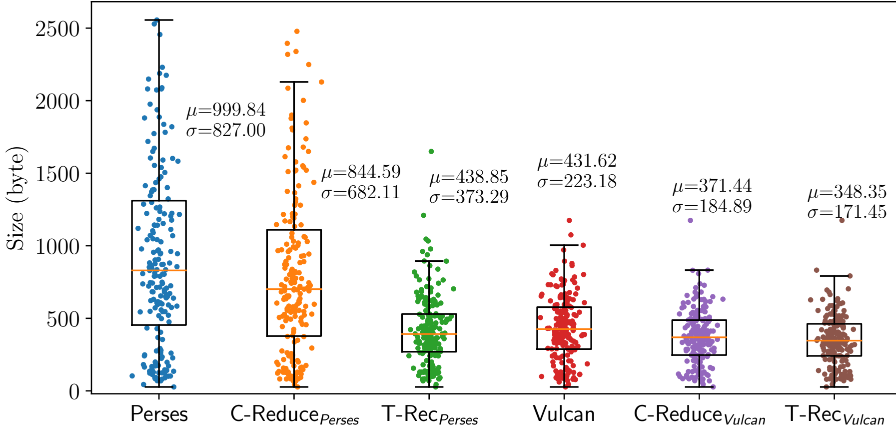

Fig. 9. The byte sizes of the results produced by different reducers on SMT-LIBv2 benchmarks. On average, the results of T-Rec$_{Perses}$ are 42.86% and 35.39% smaller than Perses and C-Reduce-cano$_{Perses}$ respectively; the results of T-Rec$_{Vulcan}$ are 16.24% and 4.83% smaller than Vulcan and C-Reduce-cano$_{Vulcan}$ respectively.

simple symbols as keywords (according to the lexical syntax) and does not reduce them, while the passes of C-Reduce treat them as identifiers and replace them with smaller ones.

> **RQ2**: Both T-Rec$_{Perses}$ and T-Rec$_{Vulcan}$ exhibit significantly better capability in canonicalization than Perses and Vulcan, respectively. Specifically, on average, the results of T-Rec$_{Perses}$ (and T-Rec$_{Vulcan}$) are 65.52% (53.73%), 28.34% (19.79%), and 42.86% (16.24%) smaller than that of Perses (and Vulcan) on C, Rust, and SMT-LIBv2 benchmarks, respectively. Although integrating the passes for canonicalization in C-Reduce to Perses and Vulcan also increases the degree of canonicalization, such an approach is not as generalized as T-Rec and is significantly outperformed by T-Rec on Rust and SMT-LIBv2 benchmarks.

5.3 RQ3: Effectiveness in Removing Tokens

The evaluation results with `Benchmark-Multi` also demonstrate that T-Rec$_{Perses}$ significantly outperforms Perses and C-Reduce-cano$_{Perses}$ in terms of removing tokens. Table 4 and Fig. 10 shows the number of tokens of the results produced by each reducer.

On average, the result of T-Rec$_{Perses}$ contains 21.30%, 4.71%, and 34.35% fewer tokens than that of Perses. In 185 (20 C, 8 Rust, and 157 SMT-LIBv2) out of 235 benchmarks, T-Rec$_{Perses}$ reduces more tokens than Perses. Although C-Reduce-cano$_{Perses}$ also reduces more tokens than Perses in some benchmarks, it only achieves this in 15 (1 C and 14 SMT-LIBv2) out of 235 benchmarks, and there is no benchmark where C-Reduce-cano$_{Perses}$ reduces more tokens than T-Rec$_{Perses}$.

T-Rec$_{Vulcan}$ does not remove more tokens than Vulcan noticeably overall. Specifically, The results of T-Rec$_{Vulcan}$ on C and Rust benchmarks contain the same number of tokens as those of Vulcan. Such a result is expected to some extent since the reduction algorithm of Vulcan is much more aggressive than Perses. Compared to Perses, it contains three additional auxiliary reducers to perform transformations for exploring reduction opportunities. Nevertheless, T-Rec$_{Vulcan}$ still manages to remove more tokens in 18 SMT-LIBv2 benchmarks. Moreover, in these 18 benchmarks, the results of T-Rec$_{Vulcan}$ contain 17.70% fewer tokens on average, and in the extreme case, the

Table 4. The numbers of tokens of the results produced by different reducers on C and Rust.

Subjects	Perses	C-Reduce-cano$_{Perses}$	T-Rec$_{Perses}$	Change *w.r.t.* Perses	Vulcan	C-Reduce-cano$_{Vulcan}$	T-Rec$_{Vulcan}$	Change *w.r.t.* Vulcan
clang-22382	144	144	129	-10.42%	108	108	108	0.00%
clang-22704	78	78	71	-8.97%	62	62	62	0.00%
clang-23309	464	464	102	-78.02%	303	303	303	0.00%
clang-23353	98	98	94	-4.08%	91	91	91	0.00%
clang-25900	239	239	211	-11.72%	104	104	104	0.00%
clang-26760	120	120	74	-38.33%	56	56	56	0.00%
clang-27137	180	180	47	-73.89%	88	88	88	0.00%
clang-27747	117	117	89	-23.93%	79	79	79	0.00%
clang-31259	406	406	331	-18.47%	282	282	282	0.00%
gcc-59903	308	308	304	-1.30%	198	198	198	0.00%
gcc-60116	488	488	331	-32.17%	241	241	241	0.00%
gcc-61383	272	272	221	-18.75%	195	195	195	0.00%
gcc-61917	150	150	134	-10.67%	103	103	103	0.00%
gcc-64990	239	239	221	-7.53%	203	203	203	0.00%
gcc-65383	153	153	134	-12.42%	84	84	84	0.00%
gcc-66186	327	327	296	-9.48%	226	226	226	0.00%
gcc-66375	440	440	368	-16.36%	227	227	227	0.00%
gcc-70127	301	301	240	-20.27%	230	230	230	0.00%
gcc-70586	157	153	142	-9.55%	94	94	94	0.00%
gcc-71626	51	51	41	-19.61%	38	38	38	0.00%
Average	236.6	236.4	179	-21.30%	150.6	150.6	150.6	0.00%
rust-111502	166	166	161	-3.01%	157	157	157	0.00%
rust-112061	458	458	413	-9.83%	442	442	442	0.00%
rust-112213	736	736	647	-12.09%	635	635	635	0.00%
rust-112526	382	382	382	0.00%	338	338	338	0.00%
rust-44800	467	467	467	0.00%	284	284	284	0.00%
rust-66851	728	728	724	-0.55%	713	713	713	0.00%
rust-69039	114	114	114	0.00%	101	101	101	0.00%
rust-77002	263	263	247	-6.08%	247	247	247	0.00%
rust-77320	39	39	39	0.00%	39	39	39	0.00%
rust-77323	13	13	13	0.00%	13	13	13	0.00%
rust-77910	34	34	28	-17.65%	21	21	21	0.00%
rust-77919	74	74	74	0.00%	62	62	62	0.00%
rust-78005	102	102	102	0.00%	102	102	102	0.00%
rust-78325	28	28	28	0.00%	26	26	26	0.00%
rust-78651	17	17	17	0.00%	9	9	9	0.00%
rust-78652	56	56	56	0.00%	49	49	49	0.00%
rust-78655	26	26	26	0.00%	26	26	26	0.00%
rust-78720	72	72	72	0.00%	56	56	56	0.00%
rust-91725	174	174	105	-39.66%	105	105	105	0.00%
rust-99830	299	299	283	-5.35%	277	277	277	0.00%
Average	212.4	212.4	199.9	-4.71%	185.1	185.1	185.1	0.00%

result of T-Rec$_{Vulcan}$ contains 63.13% fewer tokens. On the other hand, C-Reduce-cano$_{Vulcan}$ cannot remove more tokens than Vulcan in any benchmark.

> **RQ3**: T-Rec$_{Perses}$ significantly outperforms Perses and C-Reduce-cano$_{Perses}$ in terms of removing tokens. On average, the result of T-Rec$_{Perses}$ contains 21.30%, 4.71%, and 34.35% fewer tokens than that of Perses. Additionally, T-Rec$_{Vulcan}$ can remove more tokens than Vulcan in 18 out of 195 SMT-LIBv2 benchmarks despite the aggressive reduction algorithm adopted by Vulcan.

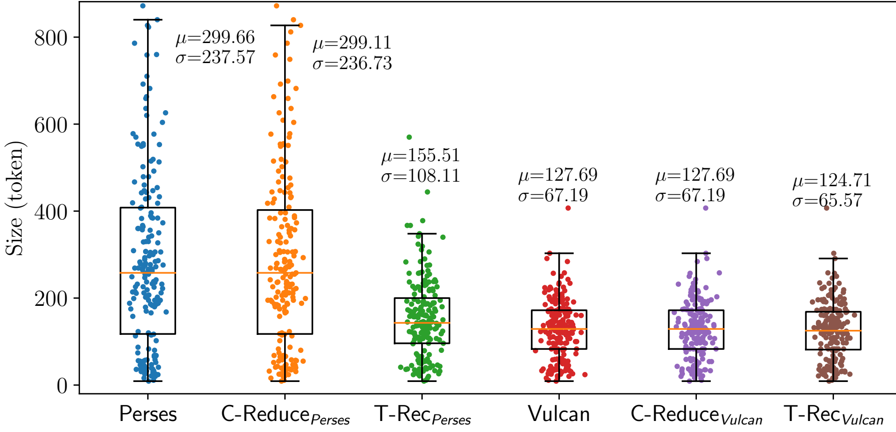

Fig. 10. The token sizes of the results produced by different reducers on SMT-LIBv2 benchmarks. On average, the results of T-Rec$_{Perses}$ contain 34.35% fewer tokens than Perses.

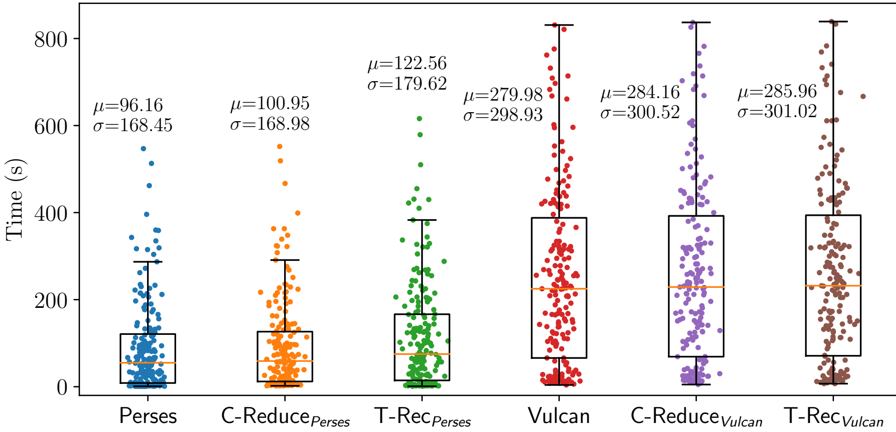

Fig. 11. The execution time of different reducers on SMT-LIBv2 benchmarks. On average, T-Rec$_{Perses}$ takes 56.46% and 22.04% more time than Perses and C-Reduce-cano$_{Perses}$ respectively; T-Rec$_{Vulcan}$ takes 5.01% more time than Vulcan.

5.4 RQ4: Efficiency

Since T-Rec$_{Perses}$ and T-Rec$_{Vulcan}$ perform extra steps on top of Perses and Vulcan, it is inevitable that T-Rec$_{Perses}$ and T-Rec$_{Vulcan}$ take more time to reduce a program. The experiment discussed in §5.1 has demonstrated this to some extent. This research question aims to further investigate how much overhead is brought by the additional fine-grained reduction process in each variant of T-Rec using `Benchmark-Multi`. To answer this research question, we compare the execution time that each tool takes to reduce the benchmarks. The result is shown in Table 5 and Fig. 11.

Perses and Vulcan as the Baselines. Overall, T-Rec$_{Perses}$ takes 24.22%, 31.22%, and 56.46% more time than Perses on C, Rust, and SMT-LIBv2 benchmarks, respectively. Compared to Vulcan, which

Table 5. The execution time of different reducers on C and Rust.

Subjects	Perses	C-Reduce-cano$_{Perses}$	T-Rec$_{Perses}$	Change *w.r.t.* Perses	Chance *w.r.t.* C-Reduce-cano$_{Perses}$	Vulcan	C-Reduce-cano$_{Vulcan}$	T-Rec$_{Vulcan}$	Change *w.r.t.* Vulcan	Change *w.r.t.* C-Reduce-cano$_{Vulcan}$
clang-22382	256	267	266	3.91%	-0.37%	525	539	534	1.71%	-0.93%
clang-22704	609	617	630	3.45%	2.11%	726	738	736	1.38%	-0.27%
clang-23309	882	964	934	5.90%	-3.11%	5,012	5,113	5,036	0.48%	-1.51%
clang-23353	320	330	329	2.81%	-0.30%	451	462	452	0.22%	-2.16%
clang-25900	388	411	429	10.57%	4.38%	848	866	859	1.30%	-0.81%
clang-26760	806	823	842	4.47%	2.31%	1,136	1,150	1,153	1.50%	0.26%
clang-27137	4,023	4,152	4,122	2.46%	-0.72%	7,027	7,148	7,085	0.83%	-0.88%
clang-27747	509	531	557	9.43%	4.90%	869	892	879	1.15%	-1.46%
clang-31259	1,013	1,343	1,927	90.23%	43.48%	9,553	9,780	9,639	0.90%	-1.44%
gcc-59903	1,822	1,872	1,898	4.17%	1.39%	2,677	2,718	2,695	0.67%	-0.85%
gcc-60116	1,252	1,442	1,718	37.22%	19.14%	6,126	6,208	6,161	0.57%	-0.76%
gcc-61383	1,670	1,876	2,141	28.20%	14.13%	9,766	9,986	9,859	0.95%	-1.27%
gcc-61917	559	572	602	7.69%	5.24%	807	819	815	0.99%	-0.49%
gcc-64990	1,502	1,553	1,798	19.71%	15.78%	3,003	3,077	3,126	4.10%	1.59%
gcc-65383	522	571	594	13.79%	4.03%	1,013	1,073	1,036	2.27%	-3.45%
gcc-66186	1,674	1,997	2,521	50.60%	26.24%	15,942	16,305	16,560	3.88%	1.56%
gcc-66375	1,763	2,097	3,344	89.68%	59.47%	10,432	10,699	10,558	1.21%	-1.32%
gcc-70127	1,949	2,310	2,969	52.33%	28.53%	13,632	13,938	13,818	1.36%	-0.86%
gcc-70586	3,317	3,830	4,214	27.04%	10.03%	15,143	15,893	15,348	1.35%	-3.43%
gcc-71626	24	26	29	20.83%	11.54%	40	43	42	5.00%	-2.33%
Average	1,243	1,379.2	1,593.2	24.22%	12.41%	5,236.4	5,372.35	5,319.55	1.59%	-1.04%
rust-111502	39	63	49	25.64%	-22.22%	275	292	283	2.91%	-3.08%
rust-112061	2,010	3,305	3,418	70.05%	3.42%	12,565	13,834	13,605	8.28%	-1.66%
rust-112213	3,235	5,287	3,649	12.80%	-30.98%	36,784	38,757	37,682	2.44%	-2.77%
rust-112526	2,643	3,112	5,064	91.60%	62.72%	7,799	8,237	9,673	24.03%	17.43%
rust-44800	456	830	575	26.10%	-30.72%	2,268	2,415	2,378	4.85%	-1.53%
rust-66851	3,170	5,500	4,771	50.50%	-13.25%	20,395	22,603	22,233	9.01%	-1.64%
rust-69039	437	584	571	30.66%	-2.23%	3,875	4,147	3,875	0.00%	-6.56%
rust-77002	191	235	189	-1.05%	-19.57%	616	651	653	6.01%	0.31%
rust-77320	7	11	7	0.00%	-36.36%	52	58	54	3.85%	-6.90%
rust-77323	4	6	4	0.00%	-33.33%	9	11	9	0.00%	-18.18%
rust-77910	7	11	9	28.57%	-18.18%	27	29	30	11.11%	3.45%
rust-77919	13	24	16	23.08%	-33.33%	96	103	100	4.17%	-2.91%
rust-78005	9	9	13	44.44%	44.44%	72	73	74	2.78%	1.37%
rust-78325	2	5	3	50.00%	-40.00%	20	21	20	0.00%	-4.76%
rust-78651	5	7	4	-20.00%	-42.86%	17	20	16	-5.88%	-20.00%
rust-78652	7	13	11	57.14%	-15.38%	58	62	57	-1.72%	-8.06%
rust-78655	2	4	3	50.00%	-25.00%	19	36	21	10.53%	-41.67%
rust-78720	12	23	16	33.33%	-30.43%	116	123	116	0.00%	-5.69%
rust-91725	649	676	783	20.65%	15.83%	1,098	1,112	1,118	1.82%	0.54%
rust-99830	6,779	7,566	8,878	30.96%	17.34%	36,539	37,060	42,597	16.58%	14.94%
Average	983.85	1,363.55	1,401.65	31.22%	-12.51%	6,135	6,482.2	6,729.7	5.04%	-4.37%

takes 265.37%, 539.89%, and 537.53% more time than Perses on C, Rust, and SMT-LIBv2 benchmarks, the overhead introduced by the fine-grained reduction process in T-Rec$_{Perses}$ is moderate, and does not make T-Rec impractical. Additionally, overhead of applying T-Rec to Vulcan is more marginal than that of applying it to Perses. Specifically, T-Rec$_{Vulcan}$ takes 1.59%, 5.04%, and 5.01% more time than Vulcan on C, Rust, and SMT-LIBv2 benchmarks, respectively.

C-Reduce-cano$_{Perses}$ and C-Reduce-cano$_{Vulcan}$ as the Baselines. Compared to C-Reduce-cano$_{Perses}$, T-Rec$_{Perses}$ takes significantly more time than C-Reduce-cano$_{Perses}$ on C and SMT-LIBv2 benchmarks (*i.e.*, 12.41% and 22.04% more time on average) and does not show significant efficiency difference on Rust benchmarks (p-value is 0.28). The reason behind this is that T-Rec is more effective in removing more tokens than C-Reduce-cano. Therefore, to reach the fixed point, more rounds of alternately invoking the syntax-guided reduction algorithm (*i.e.*, Perses) and the fine-grained reduction algorithm are required. The fact that T-Rec$_{Vulcan}$ takes a similar execution time as C-Reduce-cano$_{Vulcan}$ further verifies such an explanation.

> **RQ4**: The overhead brought by the fine-grained reduction process in T-Rec is reasonable. Specifically, T-Rec$_{Perses}$ takes 24.22%, 31.22%, and 56.46% more time than Perses on C, Rust, and SMT-LIBv2 benchmarks, respectively, and T-Rec$_{Vulcan}$ takes 1.59%, 5.04%, and 5.01% more time than Vulcan on each benchmark.

6 DISCUSSION

In this section, we discuss an ablation study we conducted to further evaluate the fine-grained reduction process of T-Rec and the potential threats to the validity of this work.

6.1 Comparing to Perses with Lex Tree Extension

A straightforward way to perform fine-grained program reduction is to first extend the parse tree of the input program with lex trees (*i.e.*, convert leaf nodes to corresponding lex trees), and then directly utilize tree-based syntax guided program reduction algorithm, *e.g.*, Perses, to reduce the extended parse tree. Compared to such an approach, the fine-grained reduction process of T-Rec contains extra canonicalization steps. As introduced in §4, T-Rec performs replacement-based token canonicalization and deletion-based token canonicalization, and deletion-based token canonicalization can be further divided into two sub-steps, *i.e.*, lexical syntax-guided reduction and lex tree-based canonicalization. Among all these steps, lexical syntax-guided reduction can be considered as directly applying the Perses algorithm to lex tree. [4] To evaluate the effectiveness of the other two extra steps that T-Rec performs, we conducted an ablation study that compares the deduplication performance of integrating T-Rec and T-Rec-reduce-only (a variant of T-Rec that only performs lexical syntax-guided reduction in the fine-grained reduction process) to Perses, Vulcan, C-Reduce, and C-Reduce-slow using the 2,501 crash test cases in `Benchmark-Tamer`. The results presented in Table 6 show that compared to reducers with T-Rec integrated (*i.e.*, T-Rec$_{Perses}$, T-Rec$_{Vulcan}$, T-Rec$_{C\text{-}Reduce}$, T-Rec$_{C\text{-}Reduce\text{-}slow}$), the corresponding reducers with T-Rec-reduce-only integrated eliminate 248, 240, 326, 223 fewer crash C test cases, indicating the extra steps performed by T-Rec significantly contributes to the high capability of canonicalization.

Table 6. The reduction results produced by different reducers on the 2,501 crash test cases in `Benchmark-Tamer`. The column **Eliminated (#)** shows the total number of programs eliminated by each reducer.

Reducer	Eliminated (#)		
	With T-Rec integrated	With T-Rec-reduce-only integrated	Difference
Perses	1,297	1,049	248
Vulcan	1,327	1,087	240
C-Reduce	1,466	1,140	326
C-Reduce-slow	2,053	1,830	223

6.2 Threats to Validity

The correctness of the implementation of T-Rec is vital to the internal validity. To mitigate the threats stemming from this aspect, we checked whether the program reduced by T-Rec could still pass the property test. The nondeterminism in some of the benchmarks is also a threat to the internal validity. Sometimes, it is hard to promise the bug is triggered in each compilation process. To mitigate this threat, first we manually patched the benchmarks that we found are nondeterministic. Specifically, we revised the corresponding property test script to repeat the test multiple times to decrease the degree of nondeterminism. Second, all the experiments except the deduplication

[4] There is still slight difference between lexical syntax-guided reduction and directly applying the Perses to the parse tree extended with lex trees. Recall that when T-Rec attempts to reduce a token, it also attempts to reduce all the other identical tokens in the same way.

experiment (§5.1) are run three times, and the results are the average over the multiple runs. For the deduplication experiment, we did not run it multiple times for all the reducers because reducers like C-Reduce-slow and T-Rec$_{\textit{C-Reduce-slow}}$ take considerable amount of time to finish. However, we did run some efficient reducers multiple times and we only observe negligible differences among different runs. Another possible threat to the validity comes from the measurement of byte size. The format of the program can affect the byte size of the program significantly. To mitigate this threat, we do not count any whitespace characters when measuring the byte size.

To mitigate the threats to the external validity, *i.e.*, the generality of T-Rec, we collected benchmarks from previous research studies [Kremer et al. 2021; Sun et al. 2018; Tian et al. 2023; Xu et al. 2023], which cover three different programming languages. Moreover, these benchmarks are diverse in terms of the size of the original bug-triggering program (ranging from 81 bytes to 793470 bytes) and bug type (crash bug and miscompilation bug).

7 RELATED WORK

In this section, we introduce and discuss some research studies that are closely related to this paper.

7.1 Language-Agnostic Program Reducers

The first work of program reduction is Delta Debugging (DD), proposed in 2002. As a ground-breaking work, DD inspires many following research studies in the field of failure-inducing input reduction [Zeller and Hildebrandt 2002]. Given a failure-inducing input, the proposed minimizing delta debugging algorithm `ddmin` systematically removes the failure-irrelevant elements in the input until removing any single element from the input cannot trigger the same failure. To improve the performance of DD, a recent study proposed Probabilistic Delta Debugging (ProbDD), which uses a probabilistic model to guide the removal of elements [Wang et al. 2021].

However, as DD treats inputs as flat lists, it can hardly handle excessively large and highly structured inputs like programs. To overcome this problem, Hierarchical Delta Debugging (HDD) converts inputs to the corresponding tree structures (*e.g.*, parse trees) and applies `ddmin` on each level of the tree in a top-down manner [Misherghi and Su 2006]. In this way, reduction starts from the coarsest granularity, gradually becomes fine-grained, and can also recognize the structure of the input; thus both the efficiency and effectiveness are enhanced. A few subsequent studies improve HDD in various ways. Picireny improves the performance of HDD by using extended context-free grammar [Hodován and Kiss 2016]. Coarse Hierarchical Delta Debugging (CHDD) significantly reduces the number of property tests performed during the reduction process without noticeably deteriorating the reduction quality [Hodován et al. 2017]. HDDr, a recursive variant of HDD [Kiss et al. 2018], significantly reduces the time required for reduction. There is another work that improves HDD by extending it with a technique called hoisting [Vince et al. 2021].

Since HDD only leverages the formal grammar to build the tree structure, it does not guarantee that the generated programs are syntactically valid. As a result, many syntactically invalid inputs are generated and tested against the property during the reduction process. Since such invalid inputs usually do not trigger the same failure as the original input, the efficiency of HDD is restricted. To address this, Sun *et al.* proposed Perses, a syntax-guided reduction approach [Sun et al. 2018]. By changing the way to use `ddmin` and designing a hoisting transformation, it avoids generating syntactically invalid inputs when trying to delete components from the program. Vulcan is a recently proposed reduction framework [Xu et al. 2023]. It contains a main reducer and utilizes a list of auxiliary reducers to help produce smaller results. Specifically, it first minimizes the program with the main reducer. Whenever the main reducer reaches a local minimum, one of the auxiliary reducers is invoked to either perform transformations or a more exhaustive search to possibly escape the local minimum, and thus the reduction can be continued by the main reducer. Tian *et al.*

further proposed *Refreshable Compact Cache*, a domain-specific caching scheme to speed up the program reduction [Tian et al. 2023].

However, all these mentioned tree-based approaches are limited by their coarse reduction granularity. All of them treat a token as an atomic element, and thus can only entirely remove a token or keep it as it is. In contrast, T-Rec can reduce tokens with a two-stage lexical syntax-guided reduction algorithm. Such a finer-grained approach enables T-Rec to achieve smaller reduction results while not introducing impractical overhead.

DDSET pursues high degree of canonicalization with a different approach [Gopinath et al. 2020a]. It strives to abstract syntactical structures of the reduced input. In other words, it attempts to convert some concrete structures in the input to generalized abstract symbols (*i.e.*, terminals and nonterminals). However, although DDSET is demonstrated to be effective with a benchmark suite including JavaScript, Clojure, Lua and UNIX command line utilities inputs, according to our evaluation, it is not as effective and efficient as T-Rec in deduplicating bug-triggering C programs. One possible explanation is that when the input has strict restrictions imposed upon its tokens and syntactical structures by the semantics or the property (*i.e.*, triggering a certain bug), the abstracting process may become infeasible, and thus the effectiveness of DDSET may significantly deteriorate.

Another test case reduction technique that is related to this work is internal test case reduction [MacIver and Donaldson 2020]. This approach also adopts shorlex order to pursuit both great reduction and canonicalization performance, but it performs reduction in a very different manner. Instead of directly performing reduction on the test case, it reduces the choice sequence that leads a test case generator to produce the test case under reduction. The advantage of internal test case reduction is that it can equip each test case generator with reduction ability for free, without the need of an external reducer. Meanwhile, it can benefit from certain properties that the generator has. For example, if a generator is crafted to only generate valid test case, then internal reduction can promise the intermediate test cases generated during the reduction process are all valid, thus avoiding wasting time on checking invalid test cases. Compared to internal test case reduction, T-Rec does not rely on the quality or presence of a test case generator, and we believe that external reduction techniques like T-Rec and internal reduction are complementary to each other.

7.2 Language-Specific Program Reducers

Besides the language-agnostic reduction approaches mentioned above, there are also many language-specific reduction tools. These tools are specifically optimized for certain languages. Therefore, they are more effective in reducing programs that are written in those specific languages. The downside of such tools is that these tools typically require domain-specific knowledge and heavy engineering efforts to build. C-Reduce is a program reduction tool specifically optimized for reducing C/C++ programs. It is implemented with a set of C/C++-specific transformations based on the Clang front end to help reduce the programs [Regehr et al. 2012]. ddSMT and ddSMT2.0 are designed for reducing SMT-LIBv2 programs [Kremer et al. 2021; Niemetz and Biere 2013]. ddSMT is based on `ddmin`, and it adds a set of SMT-LIBv2-specific structural and semantic simplifications. ddSMT2.0 further improves ddSMT by supporting the entire family of SMT-LIBv2 language dialects and providing a hierarchical reduction strategy in addition to the original `ddmin`-based strategy. JS Delta is a JavaScript-specific reducer that reduces JavaScript programs by deleting statements, functions, and sub-expressions [JS Delta 2017]. J-Reduce is a reducer for reducing Java bytecode [Kalhauge and Palsberg 2019]. It is implemented with a general strategy to overcome the challenge in reducing dependency graphs. A following work improves J-Reduce by using a more fine-grained dependency modeling approach [Kalhauge and Palsberg 2021]. CHISEL is designed for debloating C/C++ programs, which utilizes reinforcement learning to improve the performance [Heo et al. 2018].

Some of the aforementioned language-specific program reducers recognize the importance of reducing tokens to a canonical form, and are implemented with such functionality. For example, C-Reduce can refactor the program by renaming all the identifiers with shorter and canonical names (*i.e.*, `a`, `b`, `c`, $\cdots$, `aa`, `ab`, `ac`, $\cdots$), removing the content of each string literal in the program, and replacing integer tokens with shorter ones by using regular expression substitutions. However, such techniques are not generalized enough, and also they are not likely to enable further deletion of tokens. In contrast, T-Rec can canonicalize tokens for any programming language as it leverages the lexical syntax, and is demonstrated to be effective in terms of creating reduction opportunities for removing more tokens.

8 CONCLUSION

In this paper, we pointed out that the performance of previous language-agnostic program reduction approaches is restricted by their coarse reduction granularity. Because of treating tokens as the fundamental elements of a program, previous approaches can only remove an entire token or keep it as it is during reduction, but cannot shrink and canonicalize tokens. We propose T-Rec, a fine-grained language-agnostic program reduction technique. By guiding the fine-grained reduction process with the lexical syntax, T-Rec can effectively and efficiently enhance the reduction and canonicalization capability of existing language-agnostic program reducers. Our extensive evaluation with versatile benchmark suites demonstrates the effectiveness and efficiency of T-Rec. Specifically, T-Rec enables Perses, Vulcan, C-Reduce, and C-Reduce-slow to further eliminate 1,294, 1,315, 336, and 128 duplicates in a benchmark suite that contains 3,796 test cases triggering 46 bugs in GCC. The evaluation also demonstrates that T-Rec is more effective and efficient than DDSET Additionally, T-Rec further reduces 65.52% (53.73%), 28.34% (19.79%), and 42.86% (16.24%) bytes of the results of Perses (Vulcan) at the cost of 24.22% (1.59%), 31.22% (5.04%), and 56.46% (5.01%) extra execution time on C, Rust, and SMT-LIBv2 benchmarks, respectively. Moreover, T-Rec also further reduces 21.30%, 4.71%, and 34.35% tokens in the results of Perses on each language. By comparing to the approach used in C-Reduce, we also demonstrate that T-Rec has better generality and can create more reduction opportunities for removing more tokens.

ACKNOWLEDGMENTS

We sincerely thank the associated editor and anonymous reviewers of TOSEM for their constructive feedback. Their insightful suggestions significantly helped us improve this manuscript. We also would express our deepest appreciation to John Reghr, Yang Chen, Alex Groce, and Eric Eide who shared us with `Benchmark-Tamer` for conducting experiments. This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, a project under WHJIL, and CFI-JELF Project #40736.

DATA-AVAILABILITY STATEMENT

For reproducibility and replicability, we have released the implementation of T-Rec and posted the links to the benchmarks at https://github.com/trec-reducer/T-Rec.

REFERENCES

ANTLR. 2017. *The ANTLR Parser Generator*. Retrieved 2022-09-20 from https://www.antlr.org/

Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 197–208.

Nathan Chong, Alastair Donaldson, Andrei Lascu, and Christopher Lidbury. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

CPython. 2022. *Bug Report*. Retrieved 2022-09-20 from https://github.com/python/cpython/issues/new?assignees=&labels=type-bug&template=bug.md

Alastair Donaldson and David MacIver. 2021. *Test Case Reduction: Beyond Bugs*. Retrieved May 29, 2023 from https://blog.sigplan.org/2021/05/25/test-case-reduction-beyond-bugs

GCC. 2020. *A Guide to Testcase Reduction*. Retrieved 2022-10-28 from https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

GCC-Wiki. 2020. *A guide to Testcase reduction*. Retrieved 2022-09-20 from https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

Golnaz Gharachorlu and Nick Sumner. 2023. Type Batched Program Reduction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–410.

Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 2020a. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 237–248.

Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 2020b. *The DDSET Github repo*. Retrieved 2024-06-20 from https://github.com/vrthra/ddset

Alex Groce, Josie Holmes, and Kevin Kellar. 2017. One test to rule them all. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3092703.3092704

Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 380–394. https://doi.org/10.1145/3243734.3243838

Renáta Hodován and Ákos Kiss. 2016. Modernizing Hierarchical Delta Debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation* (Seattle, WA, USA) *(A-TEST 2016)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/2994291.2994296

Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse Hierarchical Delta Debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 194–203. https://doi.org/10.1109/ICSME.2017.26

Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/3278186.3278193

JerryScript. 2022. *Bug Report*. Retrieved 2022-09-20 from https://github.com/jerryscript-project/jerryscript/blob/master/.github/ISSUE_TEMPLATE/bug_report.md

JS Delta. 2017. *JS Delta*. Retrieved 2022-10-28 from https://github.com/wala/jsdelta

Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 556–566. https://doi.org/10.1145/3338906.3338956

Christian Gram Kalhauge and Jens Palsberg. 2021. Logical bytecode reduction. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1003–1016. https://doi.org/10.1145/3453483.3454091

Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: A Recursive Variant of the Hierarchical Delta Debugging Algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 16–22. https://doi.org/10.1145/3278186.3278189

Gereon Kremer, Aina Niemetz, and Mathias Preiner. 2021. ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 231–242. https://doi.org/10.1007/978-3-030-81688-9_11

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. (2014), 216–226. https://doi.org/10.1145/2594291.2594334

Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Validating JIT Compilers via Compilation Space Exploration. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 66–79. https://doi.org/10.1145/3600006.3613140

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

LLVM. 2022. *How to Submit an LLVM bug report.* Retrieved 2022-09-20 from https://llvm.org/docs/HowToSubmitABug.html

David R MacIver and Alastair F Donaldson. 2020. Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper). In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical Delta Debugging. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 142–151. https://doi.org/10.1145/1134285.1134307

Aina Niemetz and Armin Biere. 2013. ddSMT: a delta debugger for the SMT-LIB v2 format. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT*. 8–9.

John Regehr. 2019. *Design and Evolution of C-Reduce.* Retrieved 2024-01-20 from https://blog.regehr.org/archives/1679

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. https://doi.org/10.1145/3368089.3409710

Rust. 2024. *Finding a minimal, standalone example.* Retrieved 2024-01-20 from https://rustc-dev-guide.rust-lang.org/notification-groups/cleanup-crew.html#finding-a-minimal-standalone-example

Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 244–256.

Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 361–371. https://doi.org/10.1145/3180155.3180236

Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2023. On the Caching Schemes to Speed Up Program Reduction. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 17 (nov 2023), 30 pages. https://doi.org/10.1145/3617172

Dániel Vince. 2022. Iterating the minimizing Delta debugging algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation, A-TEST 2022, Singapore, Singapore, November 17-18, 2022*, Ákos Kiss, Beatriz Marín, and Mehrdad Saadatmand (Eds.). ACM, 57–60. https://doi.org/10.1145/3548659.3561314

Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. 2021. Extending hierarchical delta debugging with hoisting. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 60–69.

Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 881–892. https://doi.org/10.1145/3468264.3468625

Frank Wilcoxon. 1992. *Individual comparisons by ranking methods.* Springer.

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 718–730. https://doi.org/10.1145/3385412.3385985

Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 97 (apr 2023), 29 pages. https://doi.org/10.1145/3586049

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. PPR: Pairwise Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 338–349.