Boosting Compiler Testing via Eliminating Test Programs with Long-Execution-Time

Jiangchang Wu, Yibiao Yang*, Yuming Zhou*
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
**corresponding authors*

***Abstract*—Compiler testing is crucially important as compiler is the fundamental infrastructure in software development. One common compiler testing practice leverages a random program generator such as Csmith to generate a huge number of test programs to stress-test compilers. Each of the test programs will be compiled to different executables at different optimization levels and then their outputs will be compared against each other to differentially test compilers. However, the execution time of different test programs varies a lot. Therefore, in practice, developers often set a time limit, such as 60 or 300 seconds, to control the execution of different executables. If the execution exceeds the time limit, it will be terminated. Nevertheless, it is still unclear which time limit is more suitable for compiler testing in this context. We therefore perform the first empirical analysis to investigate how different time limits afffect the efficiency of compiler testing. We found that a time limit of 0.1 seconds can achieve the maximum benefits for compiler testing with the randomly generated test programs. At the same time, we found that 12% test programs requires more than 300 seconds for the execution and these test programs with long-execution-time (LET) consume more than 90% of the entire testing resources which makes compiler testing not cost-effectiveness. We thus propose a framework named `ELECT` to automatically identify and exclude LET test programs for boosting compiler testing. Our extensive experiments on two popular compilers GCC and LLVM have shown that `ELECT` can significantly improve the cost-effectiveness of compiler testing as it can respectively detect about 19% and 10% more bugs than the two baseline approaches under the same testing time budget. Besides, `ELECT` can respectively save about 12% and 38% time than the two other approaches for detecting the same number of bugs.**

***Index Terms*—Compiler Testing, Boosting, Long-Execution-Time, Eliminate**

I. Introduction

Compiler is the most fundamental infrastructure in software development. Ensuring the correctness of compilers is crucially important. However, compiler testing is challenging as effective test programs and test oracles are lacked. To this end, prior studies have proposed two different randomized program generators named Csmith [1] and YARPGen [2] to randomly generate test programs for testing compilers. In practice, a huge number of test programs will be randomly generated by a program generator and each of the test programs will be used as input for differentially testing compilers [3]. In particular, each test program will be first compiled by different compilers or by the same compiler with different optimization levels to obtain multiple executables. Then, the outputs of different executables will be compared against each other to manifest bugs in compilers.

In order to exhaustively test compilers, the best practice of using randomized program generator in compiler testing is to generate as many test programs as possible. Nevertheless, the execution time of different executables compiled from these randomly generated test programs vary a lot. Some of them require much more time than others for the execution or even never terminate at all. Thus, in practice, developers often set a time limit to control the execution of different executables. If an executable running out of time within a time limit, it will be directly killed automatically. Prior studies on compiler testing use different timeout settings, such as 5, 10, or 60 seconds [1], [2], [4]–[6], to control the execution of executables. However, it is still unclear which timeout setting is the best practice for compiler testing.

In this study, we thus empirically investigate how different timeout settings impact the effectiveness and efficiency of compiler testing. We first analyze the distribution of the execution time required by different test programs. We empirically found that about 88% of test programs require no more than 0.1 seconds and about 12% test programs require more than 300 seconds. Then, we compare different timeout settings to investigate which one performs better in compiler testing. Our experimental results found that setting timeout as 0.1 seconds outperforms the other settings. In other words, we do not need to wait for an executable running for 5, 10, or 60 seconds unless it terminates. In addition, we also found that 12% test programs generated by a random program generator consumes more than 90% of testing resources. Based on our investigation, we can deduce that setting a proper time limit contribute to compiler testing. Moreover, if we can identify these test programs correspond to executables that require long time for execution in advance, we can further improve the efficiency of testing by avoiding running them. In this study, we call the program corresponding to executable that requires long time for execution or even never terminating at all as a long-execution-time (LET) test program. However, it is challenging to identify LET test programs as they often contain complex semantics that result in their dynamic behaviors hard to be known statically in advance.

Motivated by that, we second propose the first technique to **E**liminate test **P**rograms with **L**ong-**E**xecution-Time for **C**ompiler **T**esting (**`ELECT`**) to identify LET test programs

This work is supported by the National Natural Science Foundation of China (62072194, 62172205) and the CCF-Huawei Populus euphratica Innovation Research Funding.

for boosting compiler testing. We observe that most of the LET test programs are lie to huge loops. Typically, each loop in the test program is correspond to multiple branches. Thus, ELECT turns to monitor the execution frequency of each branches in the test programs. More specifically, ELECT first instruments the test programs by using Gcov. This enables us to obtain the code coverage at run-time for each branch in the test program. Second, each of the instrumented test program will be compiled to obtain executable, and the executable will run for a given short period. If the executable is not terminate, ELECT will dump the code coverage of the test program. Finally, ELECT analyzes the execution frequency of each branch to determine whether there is a huge loop inside the test program. If so, ELECT identifies it as a LET test program and the test program will be excluded for testing. To evaluate the performance of ELECT, we use ELECT to filter LET test programs from the test set budget, then the rest test programs will be used to differentially test the widely-used C compilers, i.e., GCC and LLVM. Note that, we use the Different Optimization Levels (DOL) technique to differentially test compilers [7]. Our extensive experimental results show that on the one hand, more bug-revealing test programs are found. On the other hand, bugs in a compiler can be exposed earlier. For example, ELECT detects 19.18% and 10.13% more bugs than DOL using time limit 0.01 and 5 respectively for all the used compiler subjects. Moreover, ELECT spends 12.07% and 37.69% less time than DOL in the two settings on detecting the same bugs.

Contributions We make the following major contributions:

- We perform an empirical study to investigate how different timeout settings impact the effectiveness and efficiency of compiler testing.
- Our empirical study demonstrates that we could better set a small value of timeout as more bugs can be found. This enables us to execute much more test programs to better stress-testing compilers.
- We propose ELECT, the first technique for the identification of LET test programs, by running the test programs for a given short period and monitoring the execution frequency of each branches for boosting compiler testing.
- Our experimental results on GCC and LLVM demonstrating that ELECT is effective in boosting compiler testing as about 38% testing resources can be saved for exposing the same bug and about 11% bugs can be found than the best practice on timeout setting.

Paper Organization. The rest of this paper is structured as follow. Section II describes the background and our motivation. Section III describes our approach for the identification of LET test programs. Section V describes the experimental setup and devises our experimental results. Section VII is the threats to validity. Section VIII is the related works. Section IX is our conclusion and future work.

II. Background and Motivation

In this section, we first give a brief introduction to existing compiler testing techniques. Then, we empirically investigate how different timeout settings impact the effectiveness and efficiency of differential compiler testing techniques.

A. Compiler Testing Techniques

Validating compilers contains several challenging problems [8]. One of the most important challenges in compiler testing is lacking valid test programs. To this end, Yang et al. [1] proposed the well-known C program generator Csmith, a grammar-aided tool based on Randprog [9], to randomly generate C test programs for testing compilers. In particular, Csmith introduces some heuristics and safety checks to avoid undefined behaviors. A large amount of compiler bugs are found by Csmtih. However, Csmith had reached apparent saturation on the current versions of GCC and LLVM. It used heavy-handed dynamic safety checks to generate expressive random programs free of undefined behaviors [2]. Therefore, Livinskii et al. recently proposed another C program generator YARPGen, which can generate C programs free of undefined behaviors without dynamic safety checks [2]. YARPGen incorporates different static analysis techniques for code generation conservatively to avoid undefined behaviors. Besides, it implements generation policies that systematically skew probability distributions to trigger certain optimizations of the compiler.

The other important challenge in compiler testing is lacking effective test oracles. Many compiler testing techniques have been introduced to alleviate this challenge. One of the most widely-used techniques is the Randomized Differential Testing (RDT) technique [3] which tests compilers via seeking inconsistencies between multiple independently implemented compilers. In particular, the same test program will be fed to different compilers to obtain multiple executables, and then the outputs of these executables will be compared against each other. If the outputs are inconsistent, a potential bug is found in one of the compilers. Another one of the most widely-used techniques is Different Optimization Levels (DOL) which tests compilers via comparing the output of different executables compiled by the same compiler at different optimization levels [7]. Generally speaking, DOL is a variant of RDT [3]. More specifically, a given test program will first be compiled into different executables using the same compiler at different optimization levels. Then, the outputs of these executables at different optimization levels will be compared against each other to check whether they are consistent. The inconsistencies can imply a potential bug in the compiler. In addition to those, Equivalence Modulo Inputs (EMI) [10] generates a set of equivalent test programs to address the test oracle problem in compiler testing.

B. Impact of different timeout settings on compiler testing

Existing compiler testing techniques have significantly improved the reliability of compilers as they have found thousands of bugs for the most widely used compilers. Most test programs in these prior studies originated from two random program generators, Csmith [1] and YARPGen [2]. However, the execution time of those generated test programs varies a

lot. Existing studies use different timeout settings to control the execution of executables due to lacking sufficient investigation. Table I lists the timeout settings by different studies. As can be found, all these prior studies use different timeout settings to control the execution of different executables.

TABLE I
TIMEOUT USED IN PREVIOUS WORK

Timeout seconds	Tool	Approach	
5	Csmith	Generate random C programs that statically and dynamically conform to the C99 standard	[1]
10	C2V	Detect bugs in code coverage tools by randomized differential testing	[4]
60	HiCOND	Use historical data to generate more diverse program for compiler testing	[5]
300	YARPGen	Produce correct runnable C/C++ and DPC++ programs	[2]

To have a better understanding of the impact of different timeout settings for compiler testing, we first conduct a preliminary investigation on the execution time distribution on 30,000 test programs generated by Csmith 2.3.0. Csmith is the most widely-used C test-program generation tool, and it is not required to provide test inputs for these test programs generated by Csmith. Specifically, we choose GCC-4.4.0 as subject following the existing work in the field of compiler testing [7], [11]–[14] to compile these programs into executables with "-O0" option. Next, we run each executable and collect its execution time. Figure 1 shows the distribution of the execution time. We can observe that although 88% of test programs execute in less than 0.1 seconds, 12% of test programs take longer than 300 seconds to execute. We noticed that there are very few test programs with execution times between 0.1 and 300 seconds. This may be due to the insufficient diversity of test programs generated by Csmith and the small number of generated test programs. The gap may be reduced if more diverse test programs are generated. Intuitively, those LET programs can affect the efficiency of compiler testing unless an appropriate time limit is set.

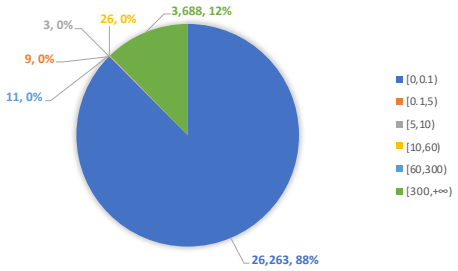

Fig. 1. Distribution of execution time of test program generated by Csmith

For further investigation, we evaluate the impact of different timeout settings on detecting bugs for GCC-4.4.0 by using the test programs generated by Csmith. We set the testing period to be 12 hours and employed the DOL testing technique. Figure 2 shows the number of bugs found for GCC-4.4.0 under different timeout settings. In general, as the value of timeout increases, fewer bugs are found. We also analyzed the time spent for each timeout to find the top-1, top-2, top-5, and top-10 bugs. The result is shown in Table II. The "$DOL_{0.1}$" column represents the time spent detecting bugs by DOL using 0.1 timeout; the same goes for the "DOL_5" column, "DOL_{10}" column, "DOL_{60}" column, and "DOL_{300}" column. $DOL_{0.1}$ spent less on detecting top-1, top-2, top-5, and top-10 bugs than DOL_5. We conducted our investigation on a workstation with an Intel 48-core 2.30GHz CPU, 120GiB RAM, and Ubuntu 18.04.3 LTS operating system. Our empirical experiments demonstrate that even the 5 seconds timeout threshold would slash the efficiency and effect of the testing process.

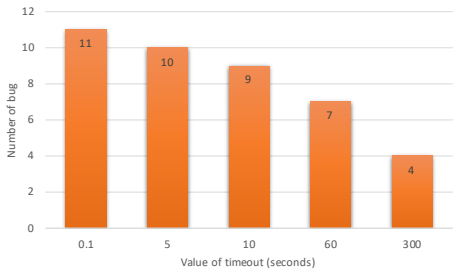

Fig. 2. Number of bugs detected by using different timeout

TABLE II
TIME SPENT ON DETECTING BUG (*10^3 SECONDS)

Bug	$DOL_{0.1}$	DOL_5	DOL_{10}	DOL_{60}	DOL_{300}
Top-1	0.25	0.26	0.27	0.37	0.85
Top-2	2.33	2.48	2.63	4.13	11.35
Top-5	15.06	16.21	17.40	29.20	—
Top-10	40.83	43.18	—	—	—

Overall, an appropriate time limit can help improve the efficiency of compiler testing. Besides, if we can accurately identify such LET test programs in advance, the testing efficiency can be further improved.

C. Illustrative Example

Figure 3 is a concrete example of a simplified LET test program generated by Csmith. In Fig. 3, line 3 defines the variable `p` with type "`short`". Line 5 is a `for` loop, `p` receives the return value from the `foo` function. However, the type of return value from the `foo` function is defined as "`unsigned char`". As the minimum value of type "`unsigned char`" is 0, `p` will be always larger than or equal to 0. As a result, the loop condition "`p >= -21`" will be always satisfied and further the `for` loop is an infinite loop. Therefore, this test program will never terminate unless running out of time. If we do not terminate the execution of this test program in time, a lot of testing resources will be wasted. We believe this kind of test program with long-execution lime will have a significantly bad effect on compiler testing. If we can identify such test programs in advance, many testing resources can be saved.

Code coverage [15] is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. It contains the execution frequencies of each line of code in the program as well as the execution percentage of each branch. To obtain code coverage of program execution by a code coverage profiler, we typically

```
1  void main ()
2  {
3    short p = 19;
4    for(p >= -21; p = foo(p, 3))
5    {...}
6  }
7
8  unsigned char foo (unsigned char a, char b)
9  {
10   return a-b;
11 }
```

Fig. 3. Huge loop generated by Csmith

need to wait for the program execution to be finished. In this study, to address this problem, we instrument the test program to obtain code coverage at runtime. We did not use the static analysis technique because the values of some variables in the test program generated by Csmtih can only be obtained when the program is running. Besides, the static analysis technique is inaccurate for the judgment of LET programs.

III. Methodology

Motivated by our observations, we propose ELECT to identify and exclude LET test program for accelerating compiler testing. Figure 4 shows the framework of ELECT. In the following, we describe the key steps in ELECT.

A. Instrument Test Program

As mentioned in Section II, the underlying reason for LET programs is that it generally contains a huge loop. To identify whether a test program is a LET test program, we turn to determine whether there is a huge loop inside. Concretely, ELECT first instruments the original test program and feeds it to the target compiler to obtain the executable. Then it executes the executable for a short period to monitor the execution frequencies of each branch at runtime. ELECT makes a decision on whether the test program is a LET one while the executable is still running. Thus, ELECT should satisfy the execution frequencies of each branch dumped at run-time. In addition, it should not change the output of the test program.

B. Filter Running Programs

In this step, ELECT will identify and filter LET test programs. Specifically, ELECT only puts an extremely short period of time to run the test program. If the executable exit in the given short period, the test program corresponding to the executable will be kept for later use in compiler testing. Otherwise, ELECT will dump the code coverage of the running program. Then, ELECT analyzes the execution frequencies of each branch in the test program to determine whether it contains any huge loop. If so, it will be considered as a LET test program and ELECT will kill the executable and filter out the test program immediately. If the test program is not a LET test program, it will also be kept to continue testing.

More significantly, it is critical for ELECT to configure an appropriate period of time threshold to execute the test program. If the period is too large, ELECT would waste a large amount of time waiting for the execution of a test program or otherwise miss many LET test programs. For the sake of convenience, we simplify this configuration problem as follows. Suppose the instrumented test programs set is $\mathscr{Q} = \{Q_1, Q_2, \ldots, Q_i\}$, where i represents the number of the test programs. Their execution time are denoted as $\mathscr{T} = \{T_1, \ldots, T_i\}$. Suppose the threshold is configured as δ, and the filter is denoted as $\mathscr{F}$.

$$\mathscr{F}(Q_i, \delta) = \begin{cases} T_i & T_i < \delta \\ \delta & T_i \geq \delta \end{cases} \tag{1}$$

Let $\mathscr{N} = \{N_1, \ldots, N_m\}$ be the set of excluded programs, where m $(0 \leq m \leq s)$ represents the number of the test program in filtered out test programs set, s is the total number of test programs. $\mathscr{R} = \{R_1, \ldots, R_n\}$ is the set of remained test programs, where n $(0 \leq n \leq s)$ represents the number of the test program in remained test program set. Suppose that the running time consumption of all the program in the filtering process is denoted as $\mathscr{C}$, it is calculated as follows:

$$\mathscr{C} = \sum_{x=0}^{m} \mathscr{F}(N_x, \delta) + \sum_{y=0}^{n} \mathscr{F}(R_y, \delta) + \xi, \ m + n = s \tag{2}$$

ξ denotes the inevitable overhead in the filtering process, which is not generated by the running of the program. Based on formula 2 and formula 1, we have the following formula:

$$\mathscr{C} = \sum_{x=0}^{n} T_x + m\delta + \xi, \ m + n = s \tag{3}$$

The values of m and n are related to the value of δ. Concretely, the larger the value of δ or n, the smaller the value of m.

To filter out a potential LET test program, the test program will be executed for a very short time. If the test program is not finished in the given short period, our tool ELECT will dump its execution coverage. Then, based on the execution coverage, ELECT will identify whether the test program is a LET test program. When ELECT identifies the test program as a LET test program, the execution of the test program will be directly killed and thus excluded from subsequent testing. Therefore, potentially large amounts of testing resources can be avoided. Intuitively speaking, if an execution is stuck in a loop, there is a high probability the loop will execute for a long time. To this end, ELECT sets a threshold γ to identify the potential huge loop to identify LET test programs. The parameter γ indicates the threshold of the execution frequency of each branch. Each test program will be executed only once. If the execution frequency of any one of the branches exceeds the threshold γ, it will be identified as a LET test program.

IV. Experimental Design

In this paper, we investigate following research questions:

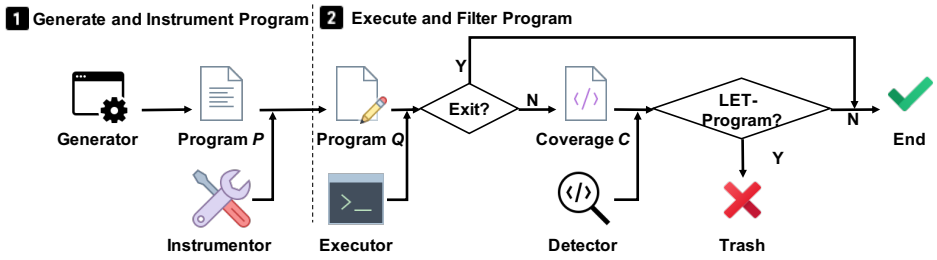

Fig. 4. The framework for ELECT

A. Research Questions

- **RQ1:** Can ELECT detect more bugs than the baselines?
- **RQ2:** Does ELECT spend less time in detecting the same bug than the baselines?
- **RQ3:** How different timeout settings in ELECT affect its ability in detecting bugs?

B. Number of Bugs Metrics

We adopted Correcting Commits [7], a method commonly used in existing studies [11]–[14], to identify the number of detected bugs from a set of failing test programs. More specifically, for any test program that triggers a bug of a compiler C whose commit version is x, the Correcting Commits method checks subsequent commits of the compiler, and determine which commits correct the bug. If two failing test programs have the same correcting commit, they are regarded as triggering the same bug. The number of correcting commits is approximately regarded as the number of detected bugs. In fact, the accuracy of this method is quite substantial as existing studies [7], [13] demonstrate.

C. Experimental Setup

In this study, all test programs we used are generated by Csmith and YARPGen [1], [2]. We choose Csmith and YARPGen as the random program generator to generate C programs to test compilers for the following main reasons: (1) they are extensively used in the literature of C compiler testing; (2) they are effective in finding bugs as thousands of bugs have been exposed and reported for the most widely-used C compilers; (3) each test case generated by Csmith and YARPGen is valid and does not require external inputs; (4) the generated programs are free from undefined behavior; and (5) they are efficient as a test program with tens of thousands of lines can be generated quickly.

Besides, we use the default option of the code generators (i.e., Csmith and YARPGen), which allows us to obtain diverse test programs at random. To have a fair comparison, we used the same random seed to control Csmith and Yarpgen for generating the same set of test programs. A unique random seed leads to a unique test program in Csmith and Yarpgen.

In line with existing compiler testing researches [1], [7], we also used two popular C compilers as subjects, i.e., GCC and LLVM. More specifically, we used two versions of GCC compilers and two versions of LLVM compilers in the x86 64-Linux platform as our subject compilers, i.e., GCC-4.3.0, GCC-4.4.0, LLVM-3.2.0, and LLVM-3.3.0. We choose these old releases rather than the latest release since: 1) they released for a long time, and thus most bugs have been exposed; 2) most bugs have been fixed as they have been maintained for a long period, thus we will have enough correcting commits to evaluate different testing strategies.

For the parameters δ and γ used in the filter and detector, we investigated the impact of these two parameters on a small test program set in Section IV-F. In addition, we set each testing period to 12 hours, and the compiler testing process runs in parallel with 25 processes during each testing period. We do not set a long testing period because this is more in line with the premise of limited test resources. To facilitate comparison, the termination time of the running test program will be set to 0.01 seconds, 0.1 seconds, and 5 seconds according to Section IV-F. They are taken as the baselines in our study. The baselines demonstrate the effectiveness of compiler testing without any accelerating approaches.

Our study was conducted on a workstation with an Intel 48-core 2.30GHz CPU, 120GiB RAM, and Ubuntu 18.04.3 LTS operating system.

D. Compiler Testing Techniques

In this study, we consider to apply ELECT to accelerate Different Optimization Level (DOL) [7], which is a representative compiler testing technique and detects a number of new bugs in compilers. We don't use EMI because EMI needs to get code coverage after the program has terminated, which is not possible for LET programs. DOL exposes compiler bugs by comparing the results produced by the same test program with different optimization levels (i.e., -O0, -O1, -Os, -O2 and -O3). If the results are inconsistent, the test program is considered to trigger a potential bug in the compiler. Given a set of test programs filtered using our approach, we compile and execute them under different optimization levels, and determine whether the test program triggers a bug by comparing their results.

E. Measurements

To measure ELECT's accuracy, we use precision and recall to evaluate ELECT's effectiveness. To find out which test programs are actually LET, we first compiled each test program with "-O0". If the compiled test program did not terminate within 300 seconds, we consider it as a truly LET test program. A higher precision is preferred as it can reveal more LET test programs during the compiler testing process.

F. Impact of Parameters

The setting may impact the effectiveness of our approach. To mitigate this potential threat, we use Csmith to generate 100,000 test programs to form a test program data set. We evaluated the impacts of δ and γ on this program set, and the experiment also conducts in parallel with 25 processes. Here we changed the value of one parameter each time and kept the values of another one unchanged. Since 88% of the test programs execute in less than 0.1 seconds, we limit the value of δ to between 0 and 0.1. To reduce the influence of random factors, we performed this experiment 10 times and calculated the average results. Figure 5 shows the impacts of the two parameters.

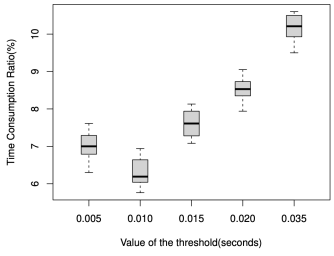

Fig. 5. Time consumption ratio

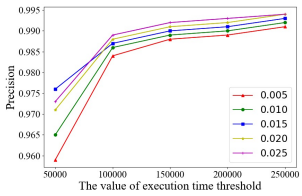

Fig. 6. The value of precision

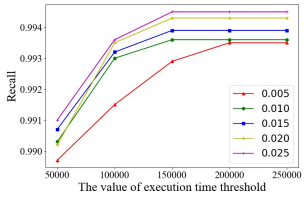

Fig. 7. The value of recall

As Figure 5 shows, when the value of δ is 0.01, the time consumption in the filtering process accounts for the smallest proportion of the total consumed time. However, the rest of the value leads to an increase in the time consumption ratio.

In fact, the value of γ is directly related to the value of δ, since the longer a LET test program is executed, the more the execution times of the branches are executed inside the program. Therefore, while discussing the value of γ, we should consider the value of δ.

From Figure 6, for all the values of δ, with the increase of the value of γ, the precision of detecting LET test program also increase. But when the value of γ exceeds 100,000, the precision changes little, and the precision of detecting LET test programs is very close regardless of the value of δ. When the value of γ is the same, and the value of δ is 0.025, the precision of detecting LET test programs is the highest. According to Figure 7, for all the values of δ, with the increase of the value of γ, the recall of detecting LET test program also increase, but when the value of γ exceeds 150,000, the recall changes little. When the value of γ is the same, and the value of δ is 0.025, the recall of detecting LET test programs is the highest.

Combined with the results in Figure 5, we choose δ as 0.01 and γ as 150,000, which can reduce the test overhead and ensure the high precision and recall of detection results.

V. Evaluation

A. Overall of Detection Result

The overall results are shown in Table III and Table IV. '$T_{0.01}$" represents that the time limit of DOL running each test program is 0.01 seconds. Similarly, "$T_{0.1}$" and "T_5" mean the time limit is 0.1 and 5 seconds respectively. A test program was considered valid if the test program terminated (correctly or otherwise) within the given seconds. "$\texttt{ELECT}_{0.01}$" and "$\texttt{ELECT}_{0.1}$" refer to using ELECT to filter out the LET test program within 0.01 and 0.1 second time limit.

1) Number of Valid Programs: Table III shows the results of valid programs during the given 12-hour testing period. According to this table, when $T_{0.01}$ is used for testing, both Csmith and YARPGen generate the greatest number of test programs. However, they generate the least number of test programs with T_5. In other words, fewer valid programs are generated as the value of the termination time increases. Regardless of the timeout value, the valid programs generated by ELECT are less than those generated by directly terminating the program with the timeout, but the number is about the same. In particular, the total number of valid programs generated by $\texttt{ELECT}_{0.01}$ (ie., 1,748,604 + 3,792,975) is 119.76% more than that of T_5 (ie., 1,145,416 + 1,376,236) and only 3.34% less than that of $T_{0.01}$ (ie., 1,787,150 + 3,945,760). The number of valid programs generated by Csmith of $\texttt{ELECT}_{0.01}$ (i.e., 1,748,604) for testing compilers is 1.53 times that of T_5 (i.e., 1,145,416) and the number of valid programs generated by YARPGen of $\texttt{ELECT}_{0.01}$ (i.e., 3,792,975) for testing compilers is 2.76 times that of T_5 (i.e., 1,376,236). The reason is that the average compilation time of the test program generated by YARPGen is longer than that of Csmith, especially when compiling with "O3". When this program is compiled with "O0", a portion of it is identified as a LET program, eliminating the need to compile it with "O3", which saves a part of the compilation time.

2) Number of Bug-revealing Programs: As is shown in the Table IV, DOL found the largest number of bug-revealing programs by using $T_{0.01}$ (i.e., 2,355 + 56). However, the number of bug-revealing programs found by $\texttt{ELECT}_{0.01}$ (i.e., 2,277 + 111) is only 0.95% less than that of $T_{0.01}$ and 19.52% more than that of T_5 (i.e., 1,826 + 172). The result from the table shows that the number of bug-revealing programs found by ELECT is almost the same as the number of bug-revealing programs found by $T_{0.01}$ and more than the number of bug-revealing programs found by T_5. Further analysis, no matter which approach is used to boost the testing process of the compiler, the test program generated by Csmith finds more bug-revealing programs than the test program generated by YARPGen. This is because the test programs generated by the two generators have different structures and are designed to detect different types of compiler bugs [2]. When using the program generated by Csmith to test the compiler, $\texttt{ELECT}_{0.01}$ finds 3.31% fewer bug-revealing programs than $T_{0.01}$, but when using the test program generated by YARPGen to test the compiler, $\texttt{ELECT}_{0.01}$ finds 98.21% more bug-revealing

TABLE III
DETECTION RESULT OF VALID PROGRAMS

Subject	Csmith					YARPGen				
	$T_{0.01}$	$T_{0.1}$	T_5	**ELECT**$_{0.01}$	**ELECT**$_{0.1}$	$T_{0.01}$	$T_{0.1}$	T_5	**ELECT**$_{0.01}$	**ELECT**$_{0.1}$
GCC-4.4.0	94,059	91,115	88,307	91,256	90,973	553,491	546,813	196,721	547,652	545,971
GCC-4.3.0	392,563	379,887	196,565	380,147	379,617	1,086,461	1,062,011	307,009	1,063,567	1,060,480
LLVM-3.3.0	609,525	599,560	394,494	599,889	599,239	1,135,376	1,073,380	446,029	1,074,389	1,072,395
LLVM-3.2.0	691,003	676,560	466,050	677,312	675,825	1,170,432	1,106,374	426,477	1,107,367	1,105,425
Total	1,787,150	1,747,122	1,145,416	1,748,604	1,745,654	3,945,760	3,788,578	1,376,236	3,792,975	3,784,271

TABLE IV
DETECTION RESULT OF BUG-REVEALING PROGRAMS AND BUGS

Generator	Subject	#Bug-revealing Programs					#Bugs				
		$T_{0.01}$	$T_{0.1}$	T_5	**ELECT**$_{0.01}$	**ELECT**$_{0.1}$	$T_{0.01}$	$T_{0.1}$	T_5	**ELECT**$_{0.01}$	**ELECT**$_{0.1}$
Csmith	GCC-4.4.0	1,427	1,383	1,337	1,386	1,380	22	26	25	26	26
	GCC-4.3.0	774	744	386	746	740	27	30	28	30	30
	LLVM-3.3.0	49	45	36	46	45	5	5	5	5	5
	LLVM-3.2.0	105	99	67	99	99	6	7	6	7	7
	Total	2,355	2,271	1,826	2,277	2,264	60	68	64	68	68
YARPGen	GCC-4.4.0	14	32	48	31	31	4	5	4	5	5
	GCC-4.3.0	15	27	39	27	27	4	6	5	6	6
	LLVM-3.3.0	12	18	30	18	18	2	3	3	3	3
	LLVM-3.2.0	15	35	55	35	35	3	5	3	5	5
	Total	56	112	172	111	111	13	19	15	19	19

programs than $T_{0.01}$. We analyzed the reason behind this phenomenon. Part of the test programs generated by YARPGen can't be terminated normally within 0.01 seconds, so directly eliminating these programs will miss a lot of bug-revealing programs. ELECT will check those test programs that haven't terminated after the program has been executed for 0.01 seconds to determine whether they are LET programs, which will preserve a large part of bug-revealing programs.

B. Number of Detected Bugs. (RQ1)

From Table IV, we counted the number of unique bugs for each subject by using correcting commits [7]. ELECT$_{0.01}$ detected the largest number of bugs among the other approaches. In particular, ELECT$_{0.01}$ detected 87 (i.e., 68 + 19) bugs, 19.18% more than $T_{0.01}$ (i.e., 60 + 13) and 10.13% more than T_5 (i.e., 64 + 15). ELECT$_{0.01}$ detected the same number of bugs as ELECT$_{0.1}$. During the same testing period, the test programs generated by Csmith can find more bugs than the test programs generated by YARPGen. For different compilers, each approach found more bugs in GCC than in LLVM.

As for the programs generated by YARPGen, even though T_5 found more bug-revealing programs than ELECT$_{0.01}$, ELECT$_{0.01}$ also found more bugs than T_5. Moreover, as for the test program generated YARPGen, although $T_{0.01}$ found more bug-revealing programs than ELECT$_{0.01}$, ELECT$_{0.01}$ also found more bugs than $T_{0.01}$. The result demonstrates that simply applying a timeout such as 0.01 and 0.1 did not have a better performance than ELECT$_{0.01}$ and ELECT$_{0.1}$, ELECT does better outperform the comparison approaches.

There is a different performance between LLVM and GCC in Table IV. We summarize the reasons as follow: (1) The release time of the LLVM subjects is about five years later than the GCC subjects. Thus, these LLVM subjects should have fewer bugs than those GCC subjects. The chosen LLVM subjects are LLVM-3.2.0 and LLVM-3.3.0, respectively, released on 1/15/2013 and 6/7/2013. While GCC-4.3.0 and GCC-4.4.0 were released in 2008 and 2009. There is a five-year gap between the LLVM subjects and the GCC subjects. In addition, Csmith was released in 2011. At that time, Csmith found 203 bugs for LLVM and 79 for GCC. In other words, most bugs found by Csmith have been fixed in those LLVM subjects as they were released two years later after Csmith. (2) Each LLVM version is maintained for a shorter period than the three subject GCC versions. LLVM-3.2.0 and LLVM-3.3.0 are only maintained for six months. While the GCC 4.3.0 and 4.4.0 are respectively maintained for 39 and 35 months. In this study, we adopted Correcting Commit to identify the number of detected bugs from a set of failing test programs. Since the investigated GCC subjects were maintained for much longer than the LLVM subjects, the GCC subjects should have much more bug-fixing commits than LLVM during the maintenance period. Note that we chose those versions of LLVM as subjects since it is hard to compile elder versions of LLVM for applying the Correcting Commits.

C. Time Spent on Detecting Each Bug. (RQ2)

The time spent on detecting each bug for each approach using Csmith is shown in Table V.The "Bug" in Table V refers to each bug, "1" refers to the first bug found by using Csmith, "2" refers to the second bug found by using Csmith, and so on. It is worth noting that each bug is unique and does not intersect with other bugs in each project. The "$T_{0.01}$" column represents the time spent on detecting each bug by "$T_{0.01}$"; the same goes for the "$T_{0.1}$" column, "T_5" column, "ELECT$_{0.01}$" column, and "ELECT$_{0.1}$" column.

TABLE V
THE TIME SPENT ON DETECTING EACH BUG BY USING CSMITH (*10^3 SECONDS)

Subject	Bug	$T_{0.01}$	$T_{0.1}$	T_5	ELECT$_{0.01}$	ELECT$_{0.1}$	Subject	Bug	$T_{0.01}$	$T_{0.1}$	T_5	ELECT$_{0.01}$	ELECT$_{0.1}$
GCC-4.4.0	1	0.00	0.00	0.00	0.00	0.00	GCC-4.3.0	23	0.00	0.00	0.00	0.00	0.00
	2	0.06	0.06	0.07	0.06	0.06		2	0.02	0.02	0.02	0.02	0.02
	3	0.10	0.10	0.11	0.10	0.10		3	0.03	0.03	0.04	0.03	0.03
	4	—	0.47	0.47	0.46	0.46		4	0.09	0.09	0.11	0.09	0.09
	5	0.71	0.71	0.73	0.71	0.71		5	0.10	0.10	0.12	0.10	0.10
	6	1.02	1.03	1.06	1.03	1.03		6	0.18	0.18	0.22	0.18	0.18
	7	—	1.19	1.22	1.19	1.19		7	0.33	0.33	0.40	0.34	0.34
	8	3.51	3.54	3.65	3.53	3.57		8	0.52	0.52	0.64	0.53	0.53
	9	6.93	6.98	7.13	6.98	7.00		9	0.72	0.73	0.90	0.73	0.73
	10	—	8.04	8.20	8.03	8.05		10	1.06	1.08	1.69	1.08	1.08
	11	11.01	11.10	11.39	11.08	11.11		11	1.14	1.16	1.94	1.16	1.17
	12	13.89	12.14	12.49	12.12	12.15		12	1.64	1.66	3.36	1.65	1.66
	13	14.35	14.40	14.81	14.38	14.41		13	1.81	1.82	3.86	1.82	1.83
	14	15.29	15.36	15.82	15.34	15.38		14	3.37	3.40	8.40	3.39	3.42
	15	17.85	17.90	18.52	17.89	17.92		15	—	3.43	8.47	3.42	3.44
	16	18.24	18.30	18.93	18.28	18.32		16	3.64	3.68	9.20	3.67	3.70
	17	20.04	20.11	20.81	20.08	20.12		17	4.34	4.39	11.21	4.37	4.41
	18	23.06	23.12	23.96	23.10	23.14		18	4.57	4.62	11.87	4.60	4.64
	19	—	23.31	24.15	23.29	23.33		19	5.49	5.55	14.59	5.54	5.58
	20	25.24	25.33	26.25	25.30	25.35		20	—	6.69	17.76	6.66	6.71
	21	26.80	26.90	27.88	26.87	26.92		21	8.20	8.26	22.22	8.23	8.30
	22	26.97	27.07	28.07	27.04	27.09		22	8.77	8.87	23.91	8.83	8.90
	23	28.64	27.97	29.02	27.95	28.00		23	10.02	10.11	27.37	10.05	10.14
	24	—	28.29	29.33	28.25	28.31		24	—	11.82	30.84	11.77	11.87
	25	30.46	30.58	31.69	30.54	30.60		25	13.13	13.26	32.54	13.20	13.31
	26	34.54	34.68	—	34.63	34.70		26	13.30	13.45	32.74	13.38	13.50
	27	43.06	—	—	—	—		27	14.34	14.50	33.99	14.42	14.55
LLVM-3.2.0	1	0.32	0.32	0.48	0.33	0.33		28	16.20	16.38	36.20	16.30	16.45
	2	0.45	0.46	0.69	0.47	0.47		29	18.89	19.07	—	18.97	19.16
	3	2.34	2.39	3.46	2.39	2.42		30	25.21	25.49	—	25.35	25.59
	4	4.50	4.57	6.64	4.57	4.63	LLVM-3.3.0	1	0.18	0.18	0.49	0.18	0.18
	5	12.80	10.01	14.44	9.93	10.09		2	0.37	0.38	0.99	0.38	0.38
	6	16.93	17.30	24.94	17.16	17.42		3	2.95	2.99	7.95	2.99	3.02
	7	—	27.90	—	27.63	28.07		4	9.32	6.83	14.60	6.79	6.88
								5	26.43	17.99	31.41	17.87	18.09

TABLE VI
THE TIME SPENT ON DETECTING EACH BUG BY USING YARPGEN (*10^3 SECONDS)

Subject	Bug	$T_{0.01}$	$T_{0.1}$	T_5	ELECT$_{0.01}$	ELECT$_{0.1}$	Subject	Bug	$T_{0.01}$	$T_{0.1}$	T_5	ELECT$_{0.01}$	ELECT$_{0.1}$
GCC-4.4.0	1	—	1.91	8.19	1.91	1.91	GCC-4.3.0	1	2.73	0.16	0.68	0.16	0.16
	2	2.59	2.63	11.20	2.63	2.64		2	2.81	0.54	2.21	0.54	0.55
	3	4.37	4.42	18.72	4.42	4.43		3	—	—	3.57	—	—
	4	7.25	7.39	31.03	7.37	7.40		4	—	0.90	3.68	0.89	0.90
	5	12.97	13.10	—	13.06	13.13		5	2.78	2.82	11.50	2.81	2.83
LLVM-3.2.0	1	0.32	0.32	1.48	0.32	0.32		6	—	5.41	—	5.39	5.44
	2	1.56	1.59	7.56	1.59	1.60		7	18.70	14.68	—	14.66	14.70
	3	—	11.87	27.63	11.81	11.91	LLVM-3.3.0	1	0.41	0.42	1.88	0.42	0.42
	4	—	12.49	—	12.43	12.54		2	—	0.38	5.79	0.38	0.38
	5	17.86	18.08	—	17.99	18.16		3	4.90	4.95	27.56	4.93	4.97

Note that since different approaches detected different numbers of bugs, some approaches may not find the bugs found by other approaches, we use "—" to align the table. For example, if "ELECT" finds the bug numbered 4 not found by "$T_{0.01}$" in Table V, then the "$T_{0.01}$" columns corresponding to the bug are marked as "—". The time spent detecting each bug for each approach using YARPGen is shown in Table VI.

We combine the data in Table V and Table VI. The result shows that $T_{0.01}$ spent less time than $T_{0.1}$ detecting the same number of bugs, and ELECT$_{0.01}$ also spent less time than $T_{0.1}$ detecting the same number of bugs. Among the common bugs (79 out of 88), both found by T_5 and ELECT$_{0.01}$, it takes less time for ELECT$_{0.01}$ to find these 79 bugs than T_5. More specifically, the total time spent by ELECT$_{0.01}$ detecting these 79 bugs is 37.69% less than that consumed by T_5 detecting these 79 bugs. There are 9 bugs found by ELECT$_{0.01}$ but not by T_5. The reason is that ELECT$_{0.01}$ generated more valid programs than T_5 to test compilers during the same testing period. There are also 1 bugs found by T_5 but not by ELECT$_{0.01}$. The reason is that ELECT$_{0.01}$ recognizes some non-LET programs as LET programs, and these non-LET programs may reveal bugs in the compilers. Interestingly, among the common bugs (67 out of 88) both found by $T_{0.01}$ and ELECT$_{0.01}$, the total time spent by ELECT$_{0.01}$ detecting these 67 bugs is 12.07% less than that spent by $T_{0.01}$ detecting these 67 bugs. The reason is that $T_{0.01}$ eliminated some programs that reveal bugs in compilers. For example, for the 5th bug found in LLVM-3.3.0 in Table V, $T_{0.01}$ spent much more time (i.e., 26.46) than ELECT$_{0.01}$ (i.e., 17.89), and so as to the 5th bug found in LLVM-3.2.0 in Table V. Among the 16 bugs not found by $T_{0.01}$, 14 were found by ELECT$_{0.01}$. The reason is that simply applying the 0.01 timeout eliminated

those test programs that triggered bugs, while $\texttt{ELECT}_{0.01}$ identified these programs as non-LET programs and preserved them. From this table, $\texttt{ELECT}_{0.01}$ boosts compiler testing in almost all cases. That demonstrates that $\texttt{ELECT}_{0.01}$ can detect almost every bug more efficiently, indicating the stably good effectiveness of $\texttt{ELECT}_{0.01}$.

D. How Do Different Value of Timeout in ELECT Affect the Effective and Efficient on Detecting Bugs? (RQ3)

We analyzed the different timeout affects on ELECT according to Table III, IV V, and VI. The results show that $\texttt{ELECT}_{0.1}$ generates fewer valid programs than $\texttt{ELECT}_{0.01}$, and $\texttt{ELECT}_{0.1}$ detects fewer bug-revealing programs than $\texttt{ELECT}_{0.01}$. Although $\texttt{ELECT}_{0.1}$ detects the same number of bugs as $\texttt{ELECT}_{0.01}$, $\texttt{ELECT}_{0.01}$ spent less time than $\texttt{ELECT}_{0.1}$ detecting each bug. It demonstrates that when the timeout value is 0.01, ELECT has a better performance than that of timeout value is 0.1.

E. ELECT's Effectiveness.

When comparing our filtering technique with the other baselines $\texttt{ELECT}_{0.01}$, $T_{0.1}$, and T_5, the extra costs incurred by instrumentation and the cost of the filtering process are all taken into account. Generally, compared to the baselines, the testing process can achieve higher efficiency with ELECT though the test programs are instrumented. To further learn whether ELECT performs better, we perform a paired t-test which can reflect the significance in statistic. The p-value between $\texttt{ELECT}_{0.01}$ and $T_{0.01}$ is 0.0006, the p-value between $\texttt{ELECT}_{0.01}$ and $T_{0.1}$ is 0.3434, and the p-value between $\texttt{ELECT}_{0.01}$ and T_5 is 0.0025. The p-value shows that $\texttt{ELECT}_{0.01}$ significantly outperforms the compared approach.

Figure 8 and Figure 9 show that ELECT achieves good precision and recall. For each subject, the precision and recall of ELECT on the test programs generated by Csmith are higher than those generated by YARPGen. Although the values of γ and δ are obtained by the test program generated by Csmith on GCC-4.4.0, they still perform well on the other subject. It demonstrates the stable effectiveness of ELECT.

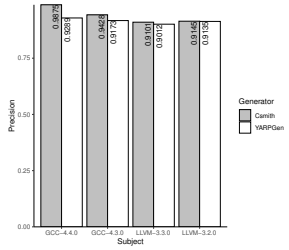

Fig. 8. The precision of ELECT

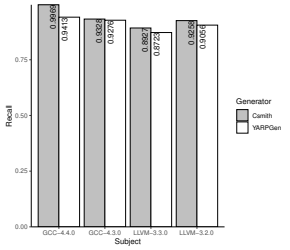

Fig. 9. The recall of ELECT

VI. Discussion

Real test programs. There may not be a better outperform if we apply the setting to real test programs. It is because our empirical study was conducted on the test programs generated by Csmith, and the execution distribution was also statistics from the test programs generated by Csmith. However, the purpose between the test programs generated by generators and the real test programs is different during the compiler testing. The generators such as Csmith and YARPGen randomly generate a vast number of programs to stress-test compilers. It has become a common practice in compiler testing, which has significantly improved the correctness of production compilers. The real test programs, such as test programs from test suites of GCC and LLVM, are limited in number. Each test program may correspond to a bug fixed in the previous version. For real test programs, we can also analyze the execution distribution of the test programs. Then we can find an appropriate threshold in a small real test program set to identify the LET test program for the real test program by the number of branch executions.

Generalization of ELECT. ELECT is also applicable to other programs, such as Java and Python. We can also find an appropriate time threshold through an empirical study based on the Java or Python programs and then identify whether a test program is a LET test program by analyzing the frequencies of branch executions.

VII. Threats to Validity

We have identified the following main threats to validity:

Impact of parameters. We used the default options of the Csmith and YARPGen, if more diverse options are adapted, we may find more bugs during testing. The values of γ and δ may impact the effectiveness of ELECT. Even though the values of γ and δ are obtained with the test program generated by Csmith running with GCC-4.4.0, ELECT still performs well on the test program generated by YARPGen and other subjects.

Impact of subjects. We used five versions of two compilers as subjects, and these subjects may not be representative enough for different compilers. We selected the two most popular C compilers to reduce this threat following the existing studies [1], [3], [7], [11], [14], [16]–[21]. More specifically, we considered different versions of different compilers to evaluate the effectiveness of ELECT from various aspects.

Impact of used techniques. We used the DOL technique in our study. However, we believe that ELECT can be generalized to various compiler testing techniques such as RDT [3]. This is because all these compiler testing techniques first utilize a test-program generator like Csmith and YARPGen to generate test programs and then use their test oracle mechanisms to detect compiler bugs. ELECT just collects code coverage during the execution of the test program to determine whether it is a LET program. Therefore, ELECT can be combined with any compiler testing technique to improve performance.

Impact of metrics. We adopted the Correcting Commits method to estimate the number of detected bugs. This method may not be precise, but it is the only metric that can automatically measure the number of bugs with some precision [7]. We implemented based on the commits on Github, and if we choose SVN, we may find more correct commits.

Different machine configurations. Our evaluations were conducted on a workstation with many cores and a large memory to eliminate the potential influence of different machine configurations. In particular, as stated in Section IV-

C, the workstation used in this study has 48 Intel 2.30 GHz cores and a memory of 120 GiB RAM. However, we only evaluate ELECT in the Linux operating system of Ubuntu 18.04.3 LTS. This may also affect our conclusions.

Limitations of ELECT. ELECT determines whether a test program is a LET test program by detecting whether it contains a huge loop. In other words, the program execution time is too long is caused by the huge loop in the program. In Section V-E, we verified the accuracy of identifying the huge loop contained in the test program by the frequencies of branch executions. We do not rule out that after waiting for an indefinite time, the program will be completed and trigger compiler bugs. However, our focus is that under the premise of limited testing resources, we can execute as many testing programs as possible so that we are more likely to find more bugs in the compiler. It is verified in Section V that during the same testing period, more compiler bugs can be found by filtering out the LET test program.

VIII. Related Work

This section introduces the related work on compiler testing techniques, compiler testing boosting technique, and techniques for huge loop detection.

A. Compiler Testing Techniques

Compiler testing is the main technology to verify the correctness of compilers. To address the test oracle problem, McKeeman et al. [3] proposed differential testing to detect bugs by checking inconsistent behaviors across comparable software or software versions. Randomized differential testing is a widely-used black-box differential testing technique in which the inputs are randomly generated [1], [22]. Yang et al. [1] proposed and implemented a tool named Csmith, which was used to randomly generate C programs without undefined behavior to test the C compiler. Le et al. [10] proposed EMI to generate some equivalent variants for each original C program, which determines whether a compiler has bugs by comparing the results produced by the original program and its variants.

B. Compiler Testing Boosting Technique

In practice, compiler testing approaches are inefficient as they require a huge cost for running a large nubmer of test programs to find a relatively small number of bugs [1], [16], [23]. To address this efficiency problem, some boosting approaches for compiler testing have been proposed. These boosting approaches can be divided into two types: test-program prioritization [11]–[13] and test-suite reduction [20]. To prioritize test programs, Chen et al. [12] proposed a method of prioritizing test programs by converting each test program into a text vector. Chen et al. [11] proposed a method to extract two kinds of features from the test program to train a model to predict the execution time of the test program and another model to predict the error detection probability of the test program. Chen et al. [13] propose an approach to distinguishing test programs based on test coverage information, which is named COP. Chen et al. [14] propose the first approach COTest by considering both optimization settings and test programs to boost compiler testing. In addition to excluding redundant test programs from a test suite, Groce et al. [20] propose reducing a compiler's test suite by simplifying each test program but retaining all test programs.

C. Techniques for Huge Loop Detection

Some existing works [24]–[26] suggested using program analysis to identify huge loops. Gupta et al. [24] presented TNT, which identifies huge loops by checking for the presence of recurrent state sets that lead a loop to execute infinitely. Velroyen et al. [25] proposed identifying huge loops with a different invariant generation technique. Burnim et al. [26] developed Looper, which combines symbolic execution with Satisfiability Modulo Theories (SMT) solvers to infer and prove the non-termination arguments of the loop. Carbin et al. [27] proposed Jolt, which used a compiler to insert instrumentation into the program. Carbin et al. [28] proposed Bolt, which operates on binaries without available source code. Researchers also developed static analysis tools, which can be used in the process of program development to statically determine whether each loop in the program is terminated or not [29]–[31]. Different from existing studies, we focus on the identification of test programs with long-execution-time rather than the identification of huge loops. In addition, the test programs in our study are only used for testing compilers. Most prior studies in huge loop detection use a static approach. Our approach is dynamically running the test program for a short period and identifying potential LET test programs by analyzing the execution frequencies.

IX. Conclusion

In this study, we first conduct an empirical study to evaluate the impact of different time limits on the effectiveness and efficiency of compiler testing. We found that the time limit used in previous work seriously affects the efficiency of the testing process. Then, we introduce a new concept of Long-Execution-Time (LET) test programs in compiler testing. To our best knowledge, existing techniques do not consider LET test programs, although it seriously affects the efficiency of compiler testing. Therefore, we propose the first simple but effective technique, named ELECT, to identify LET test programs for boosting compiler testing. More specifically, ELECT will put a very short period to execute the test program and then monitor the execution frequencies for each branch in the test program. After that, ELECT analyzes execution frequencies to determine whether there is a huge loop in the test program. If a test program has a potential huge loop identified by ELECT , it will be considered a LET test program and further excluded from later compiler testing. Our evaluations on the old versions of GCC and LLVM have demonstrated that ELECT can significantly improve the efficiency of existing compiler testing techniques as bugs can be exposed earlier, and more bug-revealing test programs can be found after filtering LET test programs.

REFERENCES

[1] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[2] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with yarpgen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428264

[3] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[4] Y. Yang, Y. Zhou, H. Sun, Z. Su, Z. Zuo, L. Xu, and B. Xu, "Hunting for bugs in code coverage tools via randomized differential testing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 488–499.

[5] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 305–316.

[6] Y. Yang, Y. Jiang, Z. Zuo, Y. Wang, H. Sun, H. Lu, Y. Zhou, and B. Xu, "Automatic self-validation for code coverage profilers," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, pp. 79–90. [Online]. Available: https://doi.org/10.1109/ASE.2019.00018

[7] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180–190.

[8] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 4:1–4:36, 2020. [Online]. Available: https://doi.org/10.1145/3363562

[9] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Proceedings of the 8th ACM international conference on Embedded software*, 2008, pp. 255–264.

[10] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, pp. 216–226.

[11] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 700–711.

[12] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *2016 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 2016, pp. 266–277.

[13] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and X. Bing, "Coverage prediction for accelerating compiler testing," *IEEE Transactions on Software Engineering*, 2018.

[14] J. Chen and C. Suo, "Boosting compiler testing via compiler optimization exploration," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, aug 2022. [Online]. Available: https://doi.org/10.1145/3508362

[15] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Commun. ACM*, vol. 6, no. 2, pp. 58–63, Feb. 1963.

[16] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 386–399, 2015.

[17] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 294–305.

[18] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 203–213. [Online]. Available: https://doi.org/10.1145/2884781.2884879

[19] V. Le, C. Sun, and Z. Su, "Randomized stress-testing of link-time optimizers," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 327–337. [Online]. Available: https://doi.org/10.1145/2771783.2771785

[20] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction: delta debugging, even without bugs," *Software Testing, Verification and Reliability*, vol. 26, no. 1, pp. 40–68, 2016.

[21] A. Groce, M. A. Alipour, C. Zhang, and Y. Chen, "Cause reduction for quick testing," in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ser. ICST '14. USA: IEEE Computer Society, 2014, p. 243–252. [Online]. Available: https://doi.org/10.1109/ICST.2014.37

[22] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 621–631.

[23] V. Le, C. Sun, and Z. Su, "Randomized stress-testing of link-time optimizers," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 327–337.

[24] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu, "Proving non-termination," in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008, pp. 147–158.

[25] H. Velroyen and P. Rümmer, "Non-termination checking for imperative programs," in *International Conference on Tests and Proofs*. Springer, 2008, pp. 154–170.

[26] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen, "Looper: Lightweight detection of infinite loops at runtime," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 161–169.

[27] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard, "Detecting and escaping infinite loops with jolt," in *European Conference on Object-Oriented Programming*. Springer, 2011, pp. 609–633.

[28] M. Kling, S. Misailovic, M. Carbin, and M. Rinard, "Bolt: on-demand infinite loop escape in unmodified binaries," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 431–450, 2012.

[29] A. R. Bradley, Z. Manna, and H. B. Sipma, "Termination of polynomial programs," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2005, pp. 113–129.

[30] M. A. Colón and H. B. Sipma, "Practical methods for proving program termination," in *International Conference on Computer Aided Verification*. Springer, 2002, pp. 442–454.

[31] B. Cook, A. Podelski, and A. Rybalchenko, "Terminator: beyond safety," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 415–418.