2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678827

Faster Mutation Analysis with Fewer Processes and Smaller Overheads

Bo Wang[1,2,3*], Sirui Lu[4,5*], Yingfei Xiong[4,5§], Feng Liu[1]
[1]School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China
[2]Beijing Key Laboratory of Traffic Data Analysis and Mining, Beijing, China
[3]CAAC Key Laboratory of Intelligent Passenger Service of Civil Aviation, Beijing, China
[4]Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, China
[5]Department of Computer Science and Technology, EECS, Peking University, Beijing, China
wangbo_cs@bjtu.edu.cn, {lsrcz, xiongyf}@pku.edu.cn, fliu@bjtu.edu.cn

***Abstract*—Mutation analysis is a powerful dynamic approach that has many applications, such as measuring the quality of test suites or automatically locating faults. However, the inherent low scalability hampers its practical use. To accelerate mutation analysis, researchers propose approaches to reduce redundant executions. A family of *fork-based approaches* tries to share identical executions among mutants. Fork-based approaches carry all mutants in one process and decide whether to *fork* new child processes when reaching a mutated statement. The mutants carried by the parent process are split into groups and distributed to different processes to finish the remaining executions. However, existing fork-based approaches have two limitations: (1) the limited analysis scope on a single statement to compare and cluster mutants prevents their systems from detecting more equivalent mutants, and (2) the interpretation of the mutants and the runtime equivalence analysis introduce significant overhead.**

In this paper, we present a novel fork-based mutation analysis approach WinMut, which (1) groups mutants in a scope of mutated statements and, (2) removes redundant computations inside interpreters. WinMut not only reduces the number of invoked processes but also has a lower cost for executing a single process. Our experiments show that our approach can further accelerate mutation analysis with an average speedup of 5.57x on top of the state-of-the-art fork-based approach, AccMut.

***Index Terms*—software testing, dynamic analysis, mutation analysis, mutation testing, fork-based mutation analysis**

I. Introduction

Mutation Analysis [1], [2] is a dynamic program analysis approach based on fault seeding. To perform mutation analysis, we first make simple syntactic changes to create a set of faulty programs called mutants. Then we execute these mutants against the test suite and compare the results with the result of the original program.

Mutation analysis is originally designed for mutation testing, i.e., evaluating the capability of a test suite for revealing faults [3], [4], [5], [6], [7]. In mutation testing, the mutants are treated as seeded faults. The more mutants a test suite detects, the more effective it is. Besides evaluating test suites, mutation analysis has been applied to many software engineering problems. For example, mutation analysis is used to automatically locate faults for relieving debugging burdens [8], [9], [10], [11], [12]. Mutants can be treated as not only patches in automated program repair [13], [14], [15], but also substitutions of real-world bugs when they are hard to collect [16]. Recently some research fields, such as smart contract [17] and deep learning [18], adopt mutation analysis to enhance system quality.

Despite the promising prospect shown both in software engineering research fields and industrial applications [19], [20], mutation analysis is still limited by its scalability issues. Given a test suite with n test cases and a program with m mutants, for each test case, the standard mutation analysis must invoke m processes to execute the mutants. This procedure results in $m*n$ executions. Although some trivially equivalent mutants can be filtered during compile time [21], m could still be very large with the program size scaling up, leading to unaffordable costs in practice.

As a result, many approaches have been proposed to enhance scalability. A basic method is to reduce run-time costs by removing redundant computations. Among them, a family of *fork-based mutation analysis approaches* tries to reduce redundant executions among mutants. A fork-based approach invokes a single process to carry the execution of all mutants. Once it encounters a mutated statement, it decides whether to fork new child process(es) to carry subsets of the active mutants carried by the current process. Split-stream execution [22], [23], [24], which is an early fork-based approach, always forks child processes to carry the mutants when it executes a mutated statement for the first time. It shares the common executions of a mutant with the original process before the first mutated statement is executed. AccMut [25], the state-of-the-art approach of the family, reduces redundancies by clustering mutants from the same mutated statement which are *equivalent modulo the current state*. Concretely, AccMut starts a process representing all mutants, shares the same executions before mutated statements as split-stream execution. When the execution reaches a statement with mutants, AccMut interprets each active mutant of the statement and collects their output states. AccMut clusters the mutants

*These two authors contribute equally to the work. §Corresponding author.
This work is supported in part by the National Key Research and Development Program of China No. 2019YFE0198100, National Natural Science Foundation of China under Grant Nos. 61922003.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00042

that have the identical output state, i.e., *equivalent modulo state*, and forks a set of child processes. Each process carries the mutants in a cluster. In this way, AccMut can share the remaining executions among the mutants in a cluster.

The overall running time of fork-based mutation analysis approach can be roughly modeled as the product of the number of processes and per-process running time. However, the existing approaches are not optimal in the two aspects:

1) *(number of processes)* The existing approaches missed a lot of opportunities to share the executions of the mutants.
2) *(per-process running time)* AccMut relies on an interpreter to handle the mutations, introducing large overhead.

Let us consider the number of processes first. For example, in the following code snippet with 3 mutants (M_1, M_2, and M_3), there are extra redundancies that are not recognized by AccMut.

```
a = b + c; //M1: (b+1)+c, M2: (b++)+c
d = a + e; //M3: (a+1)+e
return d;
```

In AccMut, these mutants will be separated into 3 different child processes, because (1) M_1 and M_2 result in two states where the values of `b` are distinct, and (2) M_3 is located in a different statement. This is not the optimal solution. As only the variable `d` affects the execution henceforth, the difference in the values of `b` will not affect the remaining executions, so it should not be included in state comparison. Furthermore, after excluding `b` from the state comparison, we can find that the mutants from two different statements (i.e., M_1 and M_3) can be equivalent. More generally, mutants in a larger scope with different program states could ultimately be equivalent. A better solution for this case is to use the current process to carry the original program and M_2, and fork a child process to carry M_1 and M_3. Extending the equivalent mutation recognition scope is non-trivial as (1) we need to recognize the effective set of program variables to do the state-comparison, and (2) we need to carefully design the algorithm to avoid introducing too much extra run-time overhead.

For the per-process running time, the overhead introduced by the current approaches is dramatically significant. To manage many mutants in a single process, the existing approaches instrument code and use an interpreter to handle the mutated statements, and the overhead is getting more significant as the algorithm for identifying the equivalent mutants at run-time is getting more complex. AccMut, for example, reports that it executes 78x more statements for a process approximately. This constant overhead introduced by the interpreter cannot be ignored. We notice the sparsity of mutated statements of a single mutant in the first-order mutation analysis scenario, that is, a mutant only contains one mutated statement. Based on this, we find that for a child process with only a subset of all the mutants, the statements could be executed with different policies: interpret the statements that are affected by some mutant (i.e. the slow way) and execute other statements directly with the compiled original version (i.e. the fast way). The implementation is non-trivial when combined with extended analysis scopes. We designed a fast algorithm to analyze the set of statements that are safe to be executed with the compiled original version, and we designed the data structures and instrumenting methods to support the runtime selection of execution policies.

```
1 uint foo(int a, int b) {
2   int sum=a+b; //M1:a-b, M2:a*b, M3:a/b
3   int avg=sum/2; //M4:(--sum)/2, M5:(sum-1)/2
4   int c=bar(avg); //bar() is side effect free
5   return c;//M6:c-1
6 }
7 void test() {
8   assert(foo(5, 1)==RESULT);
9 }
```

Figure 1: A motivating example. There are 5 mutants generated on the two expressions in the function `foo`.

We implemented our approach in WinMut as an LLVM-IR based mutation analysis framework. We have evaluated WinMut on 10 large-scale, real-world C programs with more than 20 million mutants and 1149 tests. The evaluation shows WinMut further accelerates mutation analysis with a geometric average speedup of 5.57x on top of the state-of-the-art approach, AccMut. WinMut is open source and is available at the repo https://github.com/winmutase21/WinMutASE21Artifact.

II. Motivating Example

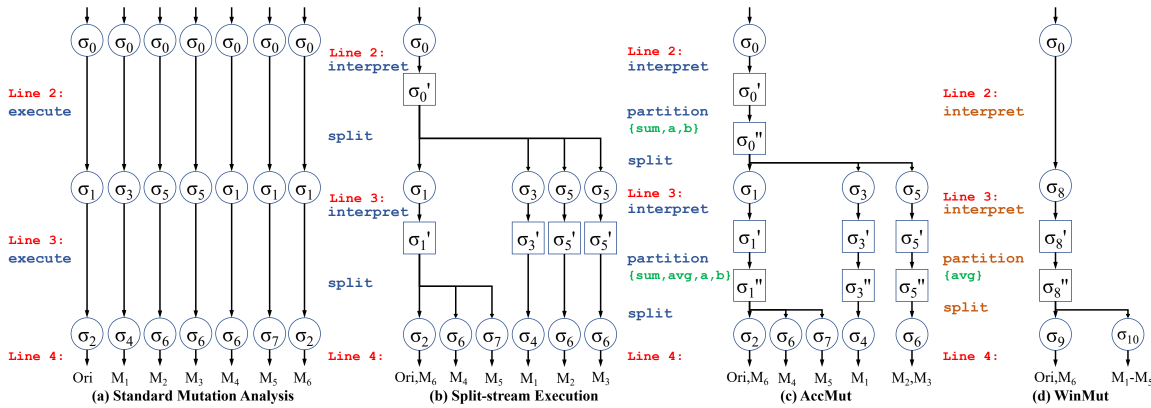
In this section, we describe the general idea of our two optimizations. We describe how a fork-based mutation analysis algorithm operates on the program in Fig. 1.

The code in Fig. 1 first calculates the average `avg` of `a` and `b` (Line 2-3), then invokes the function `bar` to get the result `c`, and finally returns `c`. To make the example simpler, we assume that there are not mutants inside of `bar`, and it is side-effect-free so that it will not change the value for `sum` and `avg`. There are 6 mutants generated on the code snippet. M_1, M_2, and M_3 change the expression at Line 2, from `a+b` to `a-b`, `a*b`, and `a/b`, respectively. M_4 and M_5 change the expression at Line 3, from `sum/2` to `(--sum)/2` and `(sum-1)/2`, respectively. M_6 changes the value to be returned at Line 5, from `c` to `c-1`.

Driven by the input, we perform standard mutation analysis, split-stream execution, AccMut, and our approach WinMut on the program respectively. We first demonstrate how these approaches execute on Line 2-4 and how WinMut reduces the number of processes. Then we show how these approaches execute on Line 5 and how WinMut avoids the overhead inside the interpreter.

A. Fewer Processes

The execution of a program can be viewed as a sequence of state transitions. Fig. 2 shows the state transitions of these approaches from the very beginning to Line 4. We use the functions mapping variables to their values to represent the states.



$\sigma_0 : \{\cdots\}$

$\sigma_1 : \{\text{sum} \mapsto 6, \cdots\}$

$\sigma_3 : \{\text{sum} \mapsto 4, \cdots\}$

$\sigma_5 : \{\text{sum} \mapsto 5, \cdots\}$

$\sigma_2 : \{\text{sum} \mapsto 6, \text{avg} \mapsto 3, \cdots\}$

$\sigma_4 : \{\text{sum} \mapsto 4, \text{avg} \mapsto 2, \cdots\}$

$\sigma_6 : \{\text{sum} \mapsto 5, \text{avg} \mapsto 2, \cdots\}$

$\sigma_7 : \{\text{sum} \mapsto 6, \text{avg} \mapsto 2, \cdots\}$

$\sigma_9 : \{\text{avg} \mapsto 3, \cdots\}$

$\sigma_{10} : \{\text{avg} \mapsto 2, \cdots\}$

$\sigma_8, \sigma_0' : \{\text{sum} \mapsto \{\{\text{Ori}, M_4, M_5, M_6\} \mapsto 6, M_1 \mapsto 4, M_2 \mapsto 5, M_3 \mapsto 5\}, \cdots\}$

$\sigma_0'' : \{(\text{sum}) \mapsto \{(6) \mapsto \{\text{Ori}, M_4, M_5, M_6\}, (4) \mapsto \{M_1\}, (5) \mapsto \{M_2, M_3\}\}, \cdots\}$

$\sigma_1' : \{\text{sum} \mapsto \{\{\text{Ori}, M_6\} \mapsto 6, M_4 \mapsto 5, M_5 \mapsto 6\}, \text{avg} \mapsto \{\{\text{Ori}, M_6\} \mapsto 3, M_4 \mapsto 2, M_5 \mapsto 2\}, \cdots\}$

$\sigma_1'' : \{(\text{sum}, \text{avg}) \mapsto \{(6, 3) \mapsto \{\text{Ori}, M_6\}, (5, 2) \mapsto \{M_4\}, (6, 2) \mapsto \{M_5\}\}, \cdots\}$

$\sigma_3' : \{\text{sum} \mapsto \{\{M_1\} \mapsto 4\}, \text{avg} \mapsto \{\{M_1\} \mapsto 2\}, \cdots\}$

$\sigma_3'' : \{(\text{sum}, \text{avg}) \mapsto \{(4, 2) \mapsto \{M_1\}\}, \cdots\}$

$\sigma_5' : \{\text{sum} \mapsto \{\{M_3\} \mapsto 5\}, \text{avg} \mapsto \{\{M_3\} \mapsto 2\}, \cdots\}$

$\sigma_5'' : \{(\text{sum}, \text{avg}) \mapsto \{(5, 2) \mapsto \{M_1\}\}, \cdots\}$

$\sigma_8' : \{\text{avg} \mapsto \{\{\text{Ori}, M_6\} \mapsto 3, M_1 \mapsto 2, M_2 \mapsto 2, M_3 \mapsto 2, M_4 \mapsto 2, M_5 \mapsto 2\}, \cdots\}$

$\sigma_8'' : \{(\text{avg}) \mapsto \{(3) \mapsto \{\text{Ori}, M_6\}, (2) \mapsto \{M_1, M_2, M_3, M_4, M_5\}\}, \cdots\}$

(e) Program States (The variables after `interpret` but before `partition` are illustrated as a mapping from mutants to values. Some mutants are grouped because (1) in split-stream execution or AccMut, they are not generated on the current statement (2) in WinMut, they are handled with specialized data structures which we will elaborate later. After `partition`, the partitioned states are illustrated as tuples of variables, and the values are illustrated as mappings from tuples of values to sets of mutants). Such as `a` and `b`, some unnecessary variables omitted in the states.

Figure 2: The procedure of standard mutation analysis, split-stream execution, AccMut, and WinMut on the code in Fig. 1. The procedure of handling a statement is decomposed into primitives. The circles represent the states after handling each statement. The squares represent the states after the primitives. States with the same labels are identical.

In the figure, the arrows stand for the transitions of states. We abstract the execution of fork-based mutation analysis with an execution engine. The execution engine can be viewed as a virtual machine and is transparent to the program. The execution engine could execute the statements by just delegating to the physical machine (the fast path), or interpret the statements with the internal states stored in the engine (the slow path). For readability, we use circles to represent states at the statement level (i.e., the states after the execution engine fully executed a statement) and use squares to represent internal states between primitives inside an execution engine (which is transparent to the program). In the figure, the states having the same name are identical. In other words, having them in multiple processes is redundant and is a waste of computing resources.

Fig. 2(a) represents the procedure of standard mutation analysis. To collect the results of the 6 mutants (M_1-M_6) and the original program (Ori), standard mutation analysis separately compiles 7 programs and runs them against the test suite in a brute force fashion. As we can see, there are considerable redundant state transitions. For example, 6 transitions to σ_0 before Line 2 and Line 3 transitions from σ_6 to the end in the processes of M_2-M_4 are redundant.

To remove redundant executions, fork-based approaches are proposed to execute multiple mutants in a single process and split the execution stream into child processes when necessary. To execute a batch of mutated statements in one process, the execution engines of fork-based approaches support a more complex design of state. In standard mutation analysis, a state maps a variable to a value, for example, $\sigma_1(\text{sum}) \to 6$. While in fork-based approaches, a state maps a variable to a function which maps a set of mutants to a value, which we call a *multi-value variable*. For example, in the state σ_0' in Fig. 2(b), `sum` is mapped to the function which consists of 4 mappings (i.e. $\{\{\text{Ori}, M_4, M_5\} \mapsto 6, \{M_1\} \mapsto 4, \{M_2\} \mapsto 5, \{M_3\} \mapsto 5\}$), and the value of `sum` for the mutant M_3 is $\sigma_0'(\text{sum})(M_3) \to 5$.

Equipped with multi-value variables, we abstract conceptually atomic operations used by fork-based mutation analysis into the following 4 primitives.

1) `execute` for delegating to the physical machine to execute the original statement,
2) `interpret` for executing a set of *active* (carried by the current process) mutated statements of the same location, and updates the states with multi-value variables,

3) `partition` for partitioning the mutants into equivalence classes by the states of multi-value variables,
4) `split` for splitting the execution stream into child processes to finish remainder executions.

Fig. 2(b) represents the procedure of split-stream execution. In split-stream execution, the execution engine starts a main process carrying all 6 mutants (M_1-M_6) and the original program (Ori). When the main process reaches the mutated statement Line 2, the execution engine enters the interpreter and invokes `interpret` to evaluate 4 versions of Line 2, which transforms the state to an internal state σ_0'. The engine performs `split` to fork 3 child processes for each mutant and continues the execution of the main process. The main process similarly proceeds Line 3, where 2 more child processes are forked for M_4 and M_5 respectively. Although split-stream execution significantly removes the redundant states before the first mutated statement (e.g., 6 redundant transitions to σ_0), it cannot reduce redundant states after the first mutated statement.

Fig. 2(c) represents the procedure of AccMut. It extends the split-stream execution's algorithm with the ability to merge mutants. Like split-stream execution, AccMut also carries all mutants with the main process. When the main process reaches Line 2, the execution engine interprets the mutated statements. The difference arises after interpreting, where AccMut further invokes the primitive `partition` with all accessible variables (`sum`, `a`, and `b`). In Fig. 2, the variables used to perform `partition` are marked in green color. The primitive `partition` clusters the states of these variables into equivalent classes, and then each class is split into a new child process. In this way, M_2 and M_3 can share the executions of equivalent mutants modulo the current state in a process. Similarly at Line 3, AccMut performs `partition` against the states of {`sum`, `avg`, `a`, `b`} and forks 2 child processes for M_4 and M_5 respectively.

Although AccMut saves execution effort between M_2 and M_3, it is still not good enough. (1) AccMut is unable to merge equivalent mutants generated on different locations. For example, M_4 and (M_2, M_3) are located at different statements, and they would be separated into two child processes because they have different program states after executing Line 2. However, if we further execute the mutants, we find that the 3 mutants finally step into the same state σ_6 under the given input, which implies that these mutants could be carried by one process and share the remaining execution. (2) AccMut does not analyze what program variables affect the final result, and can only merge mutants that result in an identical state. For instance, M_1 and (M_2, M_3) would step into σ_4 and σ_6 respectively. So they cannot be executed in one process in AccMut. However, referring to Fig. 1, we find that the different part in σ_4 and σ_6, the program variable `sum`, does not affect the final result. So it is safe to ignore this difference and execute the 3 mutants in a single process.

Fig. 2(d) shows the execution of our approach WinMut. Referring to the executions of AccMut, though some mutants are from different lines or in different states, they produce the

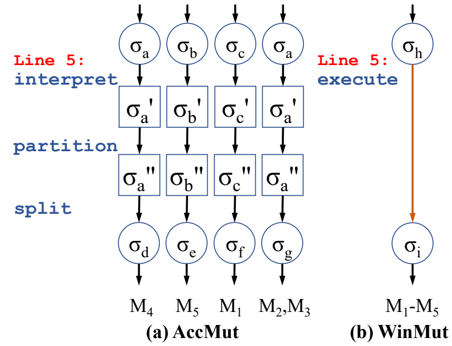


Figure 3: Redundant primitive invocations. The contents of the states $(\sigma_a, \sigma_b, \cdots)$ are not needed, so we leave them abstract.

same output after executing several statements further. Thus to merge more mutants, WinMut tries to (1) postpone the timing to perform `partition` and `split`, and (2) conservatively figure out the variables that may affect the test result to perform `partition`. WinMut continuously interprets a scope of statements (e.g., Line 2-3), until it reaches a statement that changes control-flow (e.g., the function call at Line 4), where it should perform `partition` and `split`. We decide to split at control-flow statements to avoid maintaining call stacks and path conditions, which may lead to more complex states and more significant overhead. Moreover, at Line 4, the remainder code only depends on the variable `avg`, which can be figured out by static analysis. So WinMut only clusters the active mutants against the state of the variable `avg`, which enables it to merge more mutants with possibly different program states. For example, M_4 and M_5 are clustered into a group though the values of `sum` are different ($\sigma_6(\mathrm{sum})(M_4) \to 5$, $\sigma_7(\mathrm{sum})(M_5) \to 6$). Based on our 2 innovations, M_1-M_5 are all merged into one process, which significantly reduces the number of processes.

B. Faster Processes

We compare the state-of-the-art approach AccMut with ours, to illustrate how WinMut avoids redundant invocations of high-cost primitives. Fig. 3(a) shows the executions of child processes after Line 4 in AccMut and WinMut. In the previous sub-section, AccMut forks 4 child processes carrying a set of mutants to continue the execution. When these processes reach a new mutated statement at Line 5, the execution engine still enters the interpreter, performs the sequence of high-cost primitives, that is, `interpret`, `partition` and `split`. However, by the definition of the first-order mutation analysis that each mutated program can have one mutated statement, these child processes definitely would not contain mutated statements at Line 5. In other words, these invocations of the interpreter primitives are unnecessary.

Fig. 3(b) shows how WinMut executes after Line 4 in the child process. WinMut's execution engine figures out that the child process has already carried mutants, and directly performs `execute` against Line 5. To support this, WinMut maintains a global set containing all the statements that need to be interpreted, and the set is updated when it performs `split`. More concretely, when WinMut performs `split` during interpreting Line 3, it dynamically analyzes the statements that would be affected by the active mutants (M_1-M_5), and sets the global set to these statements (Line 2-3). When the child process reaches Line 5, it finds the line does not belong to the set and executes it as is. Thus in this example, WinMut avoids the high overhead introduced by the interpreter.

This optimization is non-trivial when combined with extended analysis scopes for two reasons: (1) we need to analyze the set of statements that is safe to be delegated to the physical machine, and (2) we need to make sure that the delegation preserves the semantics. Unlike the AccMut, in which the execution engine does not hold any internal states beyond the boundary of a statement and is automatically transparent to the program, our approach needs to use specialized data structures to make the execution engine transparent and compatible with both interpreter and the physical machine.

We will elaborate on our efficient implementation later.

III. Methodology

A. Definition and Notation

In this subsection, we define a set of necessary concepts and notions which enable us to describe from an abstract view. For conciseness, we adopt some necessary definitions from the AccMut paper [25].

A program P can be viewed as a set of locations, and a mutation function p maps each location to a set of variants. Each variant v consists of a code block (denoted as $v.code$), which is either the original code block of the location or a mutated one. Let the function ori map a location to the variant containing the original code block. By the definition of first-order mutation analysis, a mutant is a program with only one mutated location. Each mutant has a unique mutation ID i. Let the function μ map a location l and a mutation ID i to a variant v, denoted as $\mu(l, i) \rightarrow v$, meaning that the mutant i should use the variant v at the location l.

Given a program P, let Σ be all the possible states and L be its location set. Let the function $\phi : \Sigma \rightarrow L$ map a state to a location of P to be executed, similar as the program counter. The execution of a program can be viewed as a sequence of state transitions, from the initial state to the terminal state $\perp$, which means the process is finished.

In standard mutation analysis, a state is a function mapping variables (i.e. storage units) to values (i.e. numbers), denoted as $\sigma : S \rightarrow \mathbb{Z}$. Here S is the universal set of the storage units of a physical machine, which includes not only the program variables in the RAM, but also the files stored in the hard drive or other resources provided by the OS. They could be handled by lazily mapping to the RAM. However, in fork-based mutation analysis, as a process may execute a set of variants at a location, a variable may come from the results of the executions of different variants. Let a multi-value variable (demoted as `MVV`) be a map from mutants to values, denoted as $\texttt{MVV} : M \rightarrow \mathbb{Z}$, where M is the universal set of the mutants of P. Thus the state in fork-based mutation analysis of a program P can be defined as a function maps variables to `MVV`s, denoted as:

$$\sigma : S \rightarrow 2^{\bigcup_{L \in P} \bigcup_{v \in p(L)} \bigcup_{z \in \mathbb{Z}} \{v \mapsto z\}}.$$

The state of standard mutation analysis is a special case of the fork-based state, whose `MVV` only contains one mapping meaning that the variable only has one value, and this mapping maps a variant set of size one.

To manipulate the multi-value variable equipped states, we define three operations: *project*, *restrict* and *update*. Given a multi-value variable equipped state σ and a mutant i, let the operation *project* return the state where all multi-value variables are reduced to single-value variables corresponding to i, denoted as $\sigma@i$. For example, given the state $\sigma : \{a \mapsto 1, b \mapsto \{Ori \mapsto 1, M_1 \mapsto 2\}\}$, we have $\sigma@M_1 : \{a \mapsto 1, b \mapsto 2\}$. As a convention, we will use ρ to denote the projected states, where all variables map to single values. Given a state σ and a variable set V, the operation *restrict* denoted by $\sigma|_V$ gets the partial state that only contains the mappings of the elements in V. The operation *update* denoted by $\sigma[o] \leftarrow v$ replaces the value of the variable o in σ to the value v, where v could be either a mapping from mutant to values or a single value.

Given a variant v and a state σ, the primitive `execute(`$v.code$`,`σ`)` executes the code block $v.code$ under the state σ and updates the state *in-place*. The function `evaluate`$(v.code, \sigma)$ evaluates v based on σ and returns the output state *without* updating the state in-place. `execute` and `evaluate` requires that all the input variables for v are single-value.

B. Models of Existing Fork-based Approaches

Based on the notations and definitions, we model the existing fork-based mutation analysis approaches in this section.

Different from the standard mutation analysis, in fork-based mutation analysis, we may execute more than one variant for the locations or execute a variant based on different program states. These locations should be *interpreted* by the execution engine. To perform interpretation, fork-based approaches invoke a procedure called `proceed` which implements their core algorithms. The main loop of fork-based mutation analysis can be modeled as Algorithm 1. First, the execution engine initializes the set G which is used to control whether a location is executed by interpretation or delegation to the physical machine, and activates all the mutants (Line 1-2). The process loops as long as there is a location to be executed (Line 3). At each step, the execution engine first picks the location to be executed (Line 4), then analyzes whether the location should be interpreted or executed by the physical machine (Line 5-9).

Split-stream execution and AccMut initialize G by calling the procedure `initialize` which adds all the mutated locations of P, and will not change G anymore. So the execution engine interprets the location by the procedure `proceed` if the current location is mutated (Line 6), otherwise delegates it to the physical machine by the primitive `execute` (Line 8).

Input: P: a program
Data: σ: the program state initialized by test input
Data: G: a set of locations should be interpreted by the current process
Data: I: a set of mutation IDs of the current process

$G \leftarrow$ initialize(P)
$I \leftarrow$ all mutant IDs of the program
while $\phi(\sigma) \neq \perp$ **do**
  $l \leftarrow \phi(\sigma)$
  if $l \in G$ **then**
    proceed(l)
  else
    execute$(ori(l).code, \sigma)$
  end
end

Algorithm 1: Main loop of fork-based mutation analysis.

In split-stream execution, the procedure `proceed` invokes the primitive `interpret` and `split` in turn. That is, it filters active mutants of the location L, then executes the code block for each mutant and records the affected values for each mutant, and finally forks new child processes for each mutant to finish the remainder executions.

AccMut optimizes the procedure `proceed` of split-stream execution by inserting an invocation of the primitive `partition` between `interpret` and `split`. The primitive `partition` builds equivalent classes based on the states of active mutants and performs `split` for each equivalent class, rather than a single mutant. In this way, AccMut merges the mutants in the equivalent classes in a process and shares their remainder executions.

As aforementioned, AccMut suffers from two limitations: (1) unable to share the executions of mutants which are from different locations or step into different states, (2) introducing considerable overhead by unnecessary entering `proceed` too many times. To overcome the limitations, we present WinMut, which reduces the number of processes and cuts down the execution overhead.

C. Fewer processes

Input: l: the current location
Data: σ: the current state
Data: I: a set of mutation IDs of the current process
Data: G: a set of locations should be interpreted by the current process
Data: CFG: the control-flow graph

1 interpret(l)
2 **if** need_split(l) **then**
3   $O \leftarrow$ output_variable(l, CFG)
4   $X \leftarrow$ partition$(\sigma|_O)$
5   split(X)
6   $pid \leftarrow$ getpid()
7   **if** is_child_process(pid) **then**
8     $G \leftarrow$ the forward slice of the locations of the mutants in I
9   **end**
10 **end**

Algorithm 2: Algorithm of `proceed` in WinMut.

The general idea of using fewer processes is to (1) enlarge the range of analysis rather than perform `partition` and `split` at each location, (2) only cluster the mutants based on a set of necessary variables (i.e. a partial state).

The procedure `proceed` of WinMut is shown in Algorithm 2. First, the execution engine performs `interpret` against the active mutants based on the current state (Line 1).

The `interpret` primitive evaluates each active mutant and maintains them as multi-value variables in the program state. If the current location is a point to perform `split` (Line 2), the execution engine filters the variables which may affect the test result by the global control-flow graph CFG (Line 3). Note that these variables can be selected by compile-time analysis, which is a sound analysis by picking out all variables that *may* affect the result. Based on the live variable set O, it performs the primitive `partition` on the partial state $\sigma|_O$ which only contains the mappings of the variables in O (Line 4). Then the execution engine groups the mutants into equivalent classes by comparing their partial states. At last, it performs the primitive `split`, for each equivalent class it forks a new child process to carry the mutants of the class and finish the remainder executions.

The scope of continuously interpreting is controlled by the procedure `need_split()`. In general, more mutants could be merged into the same equivalent class when the execution engine postpones performing `partition` and `split`. However, we can not neglect the overhead introduced by evaluating and maintaining the multi-value variables, because the primitives `interpret`, `partition`, and `split` are operated on complex data structures. This requires us to find a reasonable timing to perform `partition` and `split`. For example, if we maintain the multi-value variables in different execution paths, the multi-value should be further mapped by path conditions, which leads to unaffordable overhead. Thus we decide to partition and split when the location is a *control-flow statement*, such as branch statements and function calls. The results of need_split(l) can be statically decided during compile time.

D. Faster processes

The second improvement intends to speed up per-process execution by removing redundant interpretations. The following facts inspire us (1) a massive number of child processes are forked, (2) once a child process is split, the mutants carried by it only affect a limited range of locations that have to be interpreted, and (3) execution is much faster than interpretation.

Our basic idea is to interpret the locations that *must* be interpreted and execute other locations in child processes. Refer to the previous section, each new child process is split based on a partial state, i.e. a smaller mapping from variables to values, and we only need to interpret the locations which have active mutants and a slice of these locations.

Shown as Algorithm 1, the entrance of the interpreter is controlled by the global set G, which is initialized by the procedure initialize(P). In split-stream execution and

AccMut, G contains all the mutated locations of P in all processes. To selectively interpret, in WinMut, initialize(P) adds the forward slice locations of all mutants of P, which can be decided during compile time. The forward slice is the set of locations that depend on multi-value variables. Note that the primitive `split` converts multi-value variables to single-value ones, so the slice will not cross a split point.

Furthermore, split-stream execution and AccMut do not update G during execution. In contrast, once the primitive `split` is invoked, WinMut filters G in child processes, leaving only the locations (1) which have the active mutants of the current child process, and (2) the dynamic forward slice of these locations, shown in Line 7-9 of Algorithm 2. Because WinMut performs `split` splits at every control-flow or pointer access statement, which occurs frequently, the dynamic slice of the locations will not be so large. Consequently, G is sharply reduced to a few locations in a child process.

E. Basic Primitives

We abstract 4 necessary primitives from the operations required by fork-based mutation analysis approaches. These primitives, including `execute`, `interpret`, `partition` and `split`, are atomic operations. The `execute` primitive directly delegates a code block to the physical machine and updates the program state in place, which does not need more explanation.

Input: l: the current location
Data: I: a set of mutants IDs of the current process
Data: σ: the global program state

1 $\sigma_p \leftarrow$ an empty partial program state
2 **foreach** $i \in I$ **do**
3 $\quad \rho \leftarrow \text{evaluate}(\mu(l,i).code, \sigma@i)$
4 $\quad$ **foreach** $o \in \text{outvar}(\mu(l,i))$ **do**
5 $\quad\quad$ **if** $o \in \sigma_p.variables$ **then**
6 $\quad\quad\quad \sigma_p[o] \leftarrow \sigma_p[o] \cup \{i \mapsto \rho[o]\}$
7 $\quad\quad$ **else**
8 $\quad\quad\quad \sigma_p[o] \leftarrow \{i \mapsto \rho[o]\}$
9 $\quad\quad$ **end**
10 $\quad$ **end**
11 **end**
12 **foreach** $o \in \sigma_p$.variables **do**
13 $\quad \sigma[o] \leftarrow \sigma_p[o]$
14 **end**

Algorithm 3: The implementation of `interpret`.

The `interpret` primitive evaluates a set of mutants and updates the program state with multi-value variables, shown as Algorithm 3. For each active mutant, it first executes the variant at the current location l with mutant ID i on the projected program state $\sigma@i$ to get the result state ρ (Line 3). Then it updates the empty partial program state to ensure that $\sigma_p@i|_{\text{outvar}(\mu(l,i))} = \rho|_{\text{outvar}(\mu(l,i))}$ (Line 4-10), where outvar means the output variables of a variant. At last, it writes the variables in the partial program state back to the global program state (Line 12-14).

Algorithm 4 shows the `partition` primitive, which clusters the active mutants into equivalent classes based on the

Input: σ_p: the input partial program state
Data: I: a set of mutants IDs of the current process

1 $X \leftarrow$ empty map from projected partial program states to mutant sets
2 **foreach** $i \in I$ **do**
3 $\quad \rho \leftarrow \sigma_p@i$
4 $\quad$ **if** $\rho \in X.keyset$ **then**
5 $\quad\quad X[\rho] \leftarrow X[\rho] \cup \{i\}$
6 $\quad$ **else**
7 $\quad\quad X[\rho] \leftarrow \{i\}$
8 $\quad$ **end**
9 **end**
10 **return** X

Algorithm 4: The implementation of `partition`.

projection of the input partial state. For each mutant i, it first get the projection of the input partial state (Line 3). Then the primitive tries to find the equivalent class that i belongs to, and adds it to that class (Line 4-8). Then it returns the partition result X (Line 10).

Input: X: a map from projected partial program states to mutant sets
Data: I: the set of active mutation IDs of the current process
Data: σ: the global program state

1 **foreach** $\rho \in X$.keyset **do**
2 $\quad M \leftarrow X[\rho]$
3 $\quad pid \leftarrow$ fork()
4 $\quad$ **if** is_child_process(pid) **then**
5 $\quad\quad$ **foreach** $o \in \rho$.variables **do**
6 $\quad\quad\quad \sigma[o] \leftarrow \rho[o]$
7 $\quad\quad$ **end**
8 $\quad\quad I \leftarrow M$
9 $\quad\quad$ **return**
10 $\quad$ **end**
11 $\quad I \leftarrow (I - M)$
12 **end**

Algorithm 5: The implementation of `split`.

Algorithm 5 shows the primitive `split`, which splits executions into child process(es) for each equivalent class. For each key (i.e. a projected partial program state) in X, it gets the corresponding set of mutants M (Line 2) and forks a new process (Line 3). For the child processes, the primitive updates the variables (Line 5-7), then sets the mutants represented by the child process to M (Line 8), and returns (Line 9). For the parent process, it just removes M from the active mutants of the current process (Line 11).

Note that although the algorithms conceptually iterate through a huge set I, we can do some optimizations on this to iterate only through a subset of it and get the same result. We will elaborate on this later.

IV. IMPLEMENTATION

In this section we present WinMut implementation details. Same as AccMut, WinMut is a first-order mutation execution engine on LLVM-IR [26], that is each location contains an IR instruction. LLVM-IR is a high-level intermediate representation (IR), which is the core concept of the LLVM compiler infrastructure. IR-based mutation analysis approaches support

Table I: Mutation operators in WinMut

Name	Description	Example
AOR	Replace arithmetic operator	$a + b \rightarrow a - b$
LOR	Replace logic operator	$a \,\&\, b \rightarrow a \mid b$
ROR	Replace relational operator	$a == b \rightarrow a >= b$
LVR	Replace literal value	$T \rightarrow T + 1$
COR	Replace logical connector	$a \,\&\&\, b \rightarrow a \mid\mid b$
SOR	Replace shift operator	$a >> b \rightarrow a << b$
STDC	Delete a call	$f() \rightarrow nop$
STDS	Delete a store	$a = 5 \rightarrow nop$
UOI	Insert a unary operation	$b = a \rightarrow a++;\ b = a$
ROV	Replace the operation value	$f(a, b) \rightarrow f(b, a)$
ABV	Take absolute value	$f(a, b) \rightarrow f(abs(a), b)$

multiple front-end source languages without losing expressiveness. Particularly, LLVM-IR supports several mainstream languages, such as C/C++, Python, Objective-C, and CUDA. Recently researchers have proposed several IR-based mutation approaches, including LLVM-IR based [25], [27], [28], [29], [30], [31] and Java bytecode based [32], [15]. Note that our algorithm is general which can be applied on different code granularity, e.g., on instruction level, expression level, or statement level.

A. Mutation Operators

As each location holds an IR instruction in WinMut, we should employ IR-based mutation operators. We adopt the same set of mutation operators as AccMut, shown in Table I. These 11 mutation operators cover the mutation operators used by the state-of-the-art mutation analysis tools, such as Major [16], [33], Javalanche [32], and SRCIROR [28]. Major is a Java source code level mutation analysis tool, while Javalanche is a Java bytecode level one. All their mutation operators are employed except the Java language specified ones. SRCIROR is the state-of-art LLVM-IR based tool employing a set of 4 mutation operators, which is a subset of ours. In addition, these mutation operators are considered to be effective and are widely used in existing approaches [34], [35], [36].

B. Data Structures and Instrumentation

Although we have ensured that `execute` will not be used for an IR instruction affected by any mutation (either is mutated itself or depends on any multi-value variables), we need to ensure that (1) if an IR instruction is executed by the primitive `execute`, the delegated physical instruction could manage the multi-value variable data structure correctly, and (2) the interpretation effort should be as little as possible, which can be realized by reducing the set of mutants to be interpreted (i.e. I in Algorithm 3).

For a mutated location, we instrument the code as the following pseudo-C code:

```
if (l in G) {
  {output vars of all mutants} =
    proceed(l, {input vars of all mutants})
} else {
  {output vars} = execute(l, {input vars})
}
```

To make sure that `execute` works, we cannot change the type declarations for the variables in the original code from primitive types to mappings to support multi-value variables. Instead, we maintain the multi-value variables as two parts: *original program variable* and *additional mapping*. In the instrumented code, all of the variables are declared as the original program and always hold a single value, we call this variable the original program variable. We maintain the values in the original program variable as if they are computed with a set of `execute` calls after the last `split` primitive call.

We associate each variable with an *additional mapping* inside of the execution engine. We store those mutant/value pairs in it for those mutants with different values from the original program variable. A good property of this two-part multi-value variable data structure is that we can treat single-value variables and multi-value variables in a unified way. A single-value variable would have an empty mapping, while multi-value variables would have non-empty ones.

If the location `l` holds no mutants and the input variables are all single-value, the output variables of `proceed` procedure will all be single-value. The `proceed` procedure does nothing but maintaining the original program variables. So we can safely replace that `proceed` to `execute` and still keeps the multi-value data structures valid.

This can also reduce the redundant interpretation in Algorithm 3 as what `interpret` does now is just computing the additional mappings for the output variables. We do not need to compute all the values for the mutants in the set I. Those mutants that neither mutate the current location nor are presented in any additional mapping for the input variables can be skipped.

C. Transparent I/O System for Fork-based Mutation Analysis

Another contribution of WinMut is that we implemented a new I/O system that is transparent to users. Some fork-based mutation analysis tools [23], [25] rely on the `POSIX` system call `fork` to perform split executions. Although the *copy-on-write* mechanism of `fork` safely separates the virtual memory spaces between the parent process and the child process which avoids copying the physical memory whose pages are not written, it is unable to separate the I/O handlers between the processes. For example, if the child process writes a file that is inherited from the parent process, not only the file content is changed, but also the file pointer of the parent process is moved. To solve this problem, AccMut builds a memory mirror of all opened files, that is, it loads the whole file to memory once it opens a file. However, AccMut requires users to manually modify source code to replace all I/O operations with theirs, which needs considerable efforts. It also restricted the available APIs mostly to C standard I/Os. To deal with the restrictions, we implemented a memory-based I/O library that can be linked transparently to replace the I/O system.

Our library supports not only C standard I/Os to read or write a file, but also many file system operations like removing/creating the files. However, it would be infeasible for us to make the user-space library conform to POSIX standard and work all the same as the OS kernel. There are also features impossible for us to implement, e.g. we cannot support a program invoking `execve` or `fork` system calls. We tried our best to make the memory-based I/O system robust enough to make sure it will not crash too many tests and affect our main result.

We have a lot of assertions in our library code trying to detect inconsistency and unsupported features. When our tool executes a program, it would first execute the whole program under the memory-based I/O library to detect if there are any unsupported features. If any unsupported feature is detected or any assertion is violated, the tool would skip the mutation analysis on that program. This makes sure that we can successfully execute a test script if only a small portion of the programs would crash under our I/O system.

In the future, we may provide kernel support for the I/O operations to make them identical to the existing APIs, but this is beyond the scope of this paper.

V. Evaluation

We have evaluated WinMut on a set of real-world subjects, many of which are large-scale projects. We aim to empirically answer the following research questions:

RQ1 How does WinMut perform compared to the state-of-the-art approach AccMut?

RQ2 How is the contribution of each optimization in WinMut?

A. Experimental Setup

We implemented WinMut as a fork of AccMut, which is based on LLVM [26]. It is hard for us to manually modify the real-world projects to replace the I/O operations, and there are many I/O operations unhandled by AccMut in the real-world projects, so we modified AccMut and replaced the I/O handling module with our memory-based I/O library. This also makes sure that the two tools share the same setting, so we can compare the performance of the main algorithms. Just as AccMut, we have not implemented the support for some instructions required by C++ yet, so we only consider C subjects to answer the research questions.

We select the subjects by the following criteria:

(1) we only consider real-world, open-source subjects that have developer-written test suites;
(2) the target subjects can be compiled by LLVM;
(3) the application of the subjects should be diverse.

We selected 10 projects and their properties are shown in Table II. The column Loc shows the lines of code without comments and empty-lines, collected by the tool *cloc*. The column # Mut/# BB/# Split shows the number of mutants/basic-blocks/split point of the subject. A split point is a location to perform the primitive `split`. The column # Mut per Inst/Split is the average number of mutants for each instruction/scope of the locations corresponding to a split point.

These subjects contain in total more than 1.5 million lines of code, 20,203,516 mutants, 435,949 basic blocks, and 964,967 split points. On average, each instruction holds 16.3 mutants, and each split point handles 20.9 mutants.

Moreover, the subjects are from different fields. Binutils-gas is a portable assembler supplied by GNU. Coreutils is the GNU core utilities for manipulating files, shell and text. Gmp is an arithmetic library supplied by GNU. Libsodium is an encryption library. Lz4 is a lossless file compression program. Pcre2 is a regular expression parser that is compatible with Perl. Libpng is the official PNG library. Lua is an interpreter for the Lua language. Grep is a utility for searching plain-text data. Ffmpeg is a tool for video and audio.

As some of the subjects are very large, to complete the evaluation in a practical time budget, we do not evaluate the tools on the whole test suite. For each subject, we execute the original test suite for 2 seconds and record the covered ones as our activated tests. We also skipped the tests requiring unhandled operations by our transparent I/O systems. The column # Exec Tests of Table II shows the number of executed tests. Note that due to the diversity of the projects, they have very different organizations of test suites, one test case reported by the build system may correspond to many smaller test cases written in the test framework. We choose not to use the test case number reported by the building system but count the invoked program number for mutation analysis.

In total, we collect 1,149 tests in our evaluation. Note that although we only choose a subset of the tests, they can be still extremely time-consuming due to the intrinsic high cost of mutation analysis. In our experiment, the tests for libpng covered within the execution of 2 seconds would cost more than 4 days by AccMut. Moreover, in total, a 14-days run of AccMut is large enough for evaluation.

Following AccMut, to avoid the execution time influenced by process scheduling across multi-core, we serially executed tests without parallelization. In addition, we also limited the number of parallel processes to *one* for child processes. That is, each mutant in our experiment was executed serially. We ran WinMut 3 times on each subject, and record the average time. All experiments were evaluated on an Intel Core i7-7700K CPU and 64GB memory with Ubuntu 18.04 LTS.

B. Results and Discussion

1) RQ1: Comparison with the State-of-the-art: To answer the RQ1, we compared WinMut with the state-of-the-art fork-based approach AccMut in the following two aspects: (1) the overall execution time and (2) the number of invoked processes.

The results are shown in Table III. The columns T_w and T_a respectively show the overall execution time of WinMut and AccMut. The column T_a/T_w shows the speedup of WinMut over AccMut. While the columns P_w and P_a show the invoked process number of WinMut and AccMut. The column (P_w/P_a)% shows the percentage of process number of WinMut over AccMut. The averages are computed as geometric mean.

First, we analyze the results of execution time and we have the following findings:

(1) WinMut is faster than AccMut on all the subjects with an average speedup of 5.57x;
(2) WinMut achieves a speedup higher than 10x on 3 subjects, namely Gmp, Libsodium and Lz4. Especially, it has the maximum speedup of 28.88x on Gmp;

Table II: Subject programs

Name	Loc	# Exec Tests	# Mut	# BB	# Split	# Mut per Inst	# Mut per Split
Binutils-gas	299K	290	166,488	6,477	11,261	13.5	14.8
Coreutils	144K	287	400,150	11,532	19,628	20.4	7.2
Gmp	115K	30	613,595	10,774	23,225	22.3	26.4
Libsodium	45K	43	426,025	5,657	13,813	18.4	30.8
Lz4	13K	185	472,591	11,286	22,656	16.9	22.7
Pcre2	80K	33	266,399	6,900	11,722	16.7	22.7
Libpng	56K	9	282,831	8,527	15,394	15.0	18.4
Lua	16K	19	172,493	6,981	11,840	13.6	14.6
Grep	83K	207	217,399	8,406	16,144	12.9	13.5
Ffmpeg	1,032K	46	17,185,545	359,409	819,284	16.2	21.0
Total	1,584K	1,149	20,203,516	435,949	964,967	16.3	20.9

Table III: The total run time and the number of invoked processes of WinMut and AccMut

Subject	T_w	T_{o1}	T_{o2}	T_a	T_a/T_{o1}	T_a/T_{o2}	T_a/T_w	P_w	P_a	(P_w/P_a)%
Binutils-gas	1.62h	2.74h	1.75h	2.80h	1.02	1.60	1.72	1,580,925	1,695,842	93.2%
Coreutils	2.92m	2.96m	2.94m	2.97m	1.01	1.01	1.02	68,137	71,022	95.9%
Gmp	1.19h	37.10h	1.30h	34.26h	0.92	26.34	28.88	148,461	158,069	93.9%
Libsodium	3.94h	90.28h	4.54h	86.17h	0.95	18.98	21.86	313,007	336,904	92.9%
Lz4	1.94h	25.15h	2.16h	25.94h	1.03	12.01	13.40	118,287	130,351	90.7%
Pcre2	0.62h	4.91h	0.64h	4.99h	1.02	7.75	8.08	208,107	221,859	93.8%
Libpng	15.14h	108.71h	16.61h	111.60h	1.03	6.72	7.37	71,187	78,919	90.2%
Lua	10.08h	84.38h	10.28h	84.57h	1.00	8.22	8.39	358,892	377,177	95.2%
Grep	0.39h	1.19h	0.41h	1.28h	1.08	3.13	3.30	888,151	957,265	92.8%
Ffmpeg	2.25h	2.34h	2.54h	2.64h	1.13	1.04	1.17	390,729	441,240	88.6%
Total	37.21h	356.85h	40.29h	354.29h	1.02	5.17	5.57	4,145,883	4,468,648	92.7%

In the timing representation, h/m means hour/minute.

(3) WinMut has a more significant speedup on compute-intensive programs, such as arithmetic and encryption libraries.

Second, we evaluate the ability of WinMut to cluster more mutants. We can observe that:

(1) WinMut consistently employs fewer processes than AccMut on all the subjects;
(2) WinMut further reduces the number of invoked processes by 7.3% on average compared with AccMut.

2) RQ2: Contribution of Each Optimization: WinMut consists of 2 individual optimizations, i.e., the one for merging more mutants and the one for operating more efficiently. These optimizations may have different effects on the overall speedup, and this question intends to detailed evaluate their contribution. To answer this question, we conducted a controlled trial. That is, we only activate one optimization and compare the overall execution time.

Table III shows the results. The columns T_w, T_{o1}, T_{o2} and T_a show the execution time of WinMut, WinMut with the first optimization (for merging more mutants), WinMut with the second optimization (for more efficient execution), and AccMut, respectively. The column T_x/T_y means the speedup of the technique y over x.

We can make the following findings:

(1) the second optimization boosts the execution;
(2) the first optimization introduces speed reduction on the subjects Gmp and Libsodum and improve the performance slightly on the remaining subjects;
(3) except on the subject Ffmpeg, the second optimization contributes a higher speedup than the first one on all subjects;
(4) the speedup of the second optimization is closer to the final speedup of WinMut;
(5) the combination of the 2 optimizations results in a better speedup than employing just one of them.

As discussed in the previous section, merging more mutants involves heavier costs that would cover the benefits. So it is reasonable that the first optimization slightly slows down the execution on Gmp and Libsodium. Moreover, the first optimization boosts more than the second optimization on the subject Ffmpeg for it merges more mutants according to Table III. Finally, the final speedup can not be predicted by simply multiply the speedup results of the optimizations, which implies the combination of the two techniques has complex mutual influence. However, the time required by WinMut is only 92% of the time required by AccMut + second optimization, which is very close to the process number ratio. This indicates that the second optimization minimizes the impact of mutant merging algorithms and makes it possible to use a more powerful algorithm in the future research to merge more mutants without introducing too much overhead.

C. The robustness and impact of the memory-based I/O system

To verify that our memory-based I/O system will not affect the main results, we executed all test cases from the subjects and recorded how many test cases it passes.

In our experiment, we are not counting the test case number reported by the script, because each reported test case could correspond to many smaller test cases. We count how many test programs are executed with our I/O system. For the time, we only count the time for executing the subject program. We do not count the time for executing the external test framework (e.g. DejaGnu or manually written test script).

First, we execute the whole test suite without our I/O system, record the *total test cases number*. Then we execute the whole test suite with our I/O system. If the test script does not finish normally, we remove the failing test cases and those test cases depending on them and restart until the tests remaining in the test script all pass. Then we record the *process number*. We subtract this process number with the number of processes with detected unsupported features, and we get the *total passing test cases*.

Then we rerun the current test script with and without our I/O system and record the *time*. The results are shown in Table IV. The running time is comparable, indicating that our I/O library does not introduce too much overhead.

Table IV: Passed cases and running time with/without our I/O lib

	Without I/O lib		With I/O lib	
Subject	**#Cases**	**Time/s**	**#Cases**	**Time/s**
Binutils-gas	2368	16.09	2368	20.91
Coreutils	31698	8.23	16597	29.67
Gmp	170	23.00	170	22.62
Libsodium	77	4.26	77	4.04
Lz4	185	8.52	185	5.79
Pcre2	33	0.17	33	0.19
Libpng	38	65.13	38	64.86
Lua	28	3.74	21	3.83
Grep	3604	0.92	2743	3.70
Ffmpeg	4398	146.31	4388	163.37

We noticed that the library breaks some test cases in Coreutils, Lua, Grep and Ffmpeg. Coreutils is a library whose test cases are testing all kinds of edge cases and we expect that it would crash our library the most, and we found that our `unlink` implementation fails to handle some cases. Lua failed to execute some of the tests because they used `fork`. Grep and Ffmpeg are querying unsupported files. e.g., Ffmpeg is trying to read `/dev/urandom`, which is not supported.

VI. Related Work

In this section, we first present related work on accelerating mutation analysis, then we introduce related fields. Based on the survey papers [34], [35], [36], we can roughly divide existing approaches into static approaches and dynamic approaches.

Statically Accelerating Mutation Analysis. Static approaches intend to reduce the cost of mutation analysis without executing mutants against test suites. Basically, static approaches aim to reduce cost during mutation generation and compile time.

Several approaches use static analysis of compilers to remove the useless equivalent mutants [37], [21], [38] or improve the effectiveness [27]. As the costs of mutation analysis are positively associated with the number of mutants and the size of test suites, existing approaches mainly focus on reducing them.

A popular class of methods is to select a subset of the mutants, such as mutation sampling [39], mutation clustering [40], and mutation operator selection [41], [42]. Some comprehensive approaches combining several techniques [43], [44], [45], [46]. Some approaches utilize machine learning models trained by real-world bugs to prioritize the high-quality mutants [29], [47], or focus on the newly committed code [48]. Some other methods analyze test suites, such as test selection [49] and figure out the reusable test results in regression testing [50]. Some ML-based methods try to predict the results of mutants and avoid execution [51], [52]. These approaches can be pre-process filters and combined with ours.

Dynamically Accelerating Mutation Analysis. As mutation analysis is a kind of dynamic approach essentially, some existing studies aim to reduce runtime costs of mutation analysis. Besides mutant reduction techniques [53], [54], the majority of dynamic approaches focus on reducing redundant certain parts of mutation analysis.

Some approaches intend to reduce compile time redundancies. Mutant schemata [55] compiles all mutants once into a single executable file. Some incipient approaches avoid compile-time costs in an interpreting fashion [56], but they are usually lumbered by the low-efficiency of interpreters.

The prevalent dynamic method is to reduce redundancies during executing mutants. Split-stream execution [23], [22] reduces the redundant executions before the first mutated statement. Just et al. cluster mutants are test equivalent [57]. AccMut [25] as mentioned before, tries to further merge mutants of the same states. As discussed before, our approaches could outperform these approaches.

Higher-order mutation analysis [58], [59] replaces more than one statement once in a program, which is very different from standard mutation analysis, and some approaches aim to share executions in higher-order mutation analysis [24], [60]. Finally, some works resort the test cases to kill mutants faster [61], [49]. These approaches are orthogonal to ours.

Sharing Executions in Software Product-line Testing, Model Checking and Symbolic Execution. Similar ideas of sharing executions also exist in the fields of software product-line testing, model checking and symbolic execution approaches, which face the same challenge of redundancy.

A product in software product-line can be treated as a higher-order mutant [60]. Variational execution maintains a set of multi-value variable across the entire test execution to share common executions [62], [63], [64]. These approaches aim to merge products (i.e. higher-order mutants) via purely interpreting, which leads to significant overhead.

Multi-valued model checking supports variables in the finite state machine to be multi-valued [65]. Delta symbolic execution [66], shadow symbolic execution [67], and multi-path symbolic execution [68] try to share common parts of multiple paths.

VII. Conclusion

In this paper, we propose a novel approach to accelerate mutation analysis. We take the existing fork-based mutation analysis approaches a step further by (1) reducing the number of invoked processes, and (2) removing redundancies inside the execution engine. We implemented our approach into the tool WinMut. The evaluation results show that our approach achieves an average speedup of 5.57x on top of the state-of-the-art approach, AccMut.

REFERENCES

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[2] R. G. Hamlet, "Testing programs with the aid of a compiler," *Software Engineering, IEEE Transactions on*, vol. SE-3, no. 4, pp. 279–290, 1977.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[4] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 435–445.

[5] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 597–608.

[6] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, "How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2426–2463, 2018.

[7] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 537–548.

[8] M. Papadakis and Y. Le Traon, "Using mutants to locate" unknown" faults," in *ICST*, 2012, pp. 691–700.

[9] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *ICST*, 2014, pp. 153–162.

[10] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[11] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 609–620.

[12] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[13] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012, pp. 3–13.

[14] W. Weimer, Z. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *ASE*, 2013, pp. 356–366.

[15] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[16] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *FSE*, 2014, pp. 654–665.

[17] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen, "Musc: A tool for mutation testing of ethereum smart contract," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1198–1201.

[18] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1158–1161.

[19] G. Petrović and M. Ivanković, "State of mutation testing at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, pp. 163–171.

[20] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just, "An industrial application of mutation testing: Lessons, challenges, and research directions," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 47–53.

[21] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *ICSE*, 2015, pp. 936–946.

[22] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[23] R. Gopinath, C. Jensen, and A. Groce, "Topsy-Turvy: a smarter and faster parallelization of mutation analysis," in *ICSE*, 2016, pp. 740–743.

[24] S. Tokumoto, H. Yoshida, K. Sakamoto, and S. Honiden, "Muvm: Higher order mutation analysis virtual machine for c," in *ICST*, 2016, pp. 320–329.

[25] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 295–306.

[26] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.

[27] F. Hariri, A. Shi, H. Converse, S. Khurshid, and D. Marinov, "Evaluating the effects of compiler optimizations on mutation testing at the compiler ir level," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 105–115.

[28] F. Hariri and A. Shi, "Srciror: A toolset for mutation testing of c source code and llvm intermediate representation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 860–863.

[29] M. Papadakis, T. T. Chekam, and Y. Le Traon, "Mutant quality indicators," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018, pp. 32–39.

[30] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, "Comparing mutation testing at the levels of source code and compiler intermediate representation," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 114–124.

[31] T. T. Chekam, M. Papadakis, and Y. Le Traon, "Mart: a mutant generation tool for llvm," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1080–1084.

[32] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for Java," in *ESEC/FSE*, 2009, pp. 297–298.

[33] R. Just, F. Schweiggert, and G. M. Kapfhammer, "Major: An efficient and extensible tool for mutation analysis in a Java compiler," in *ASE*, 2011, pp. 612–615.

[34] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, 2011.

[35] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[36] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, p. 110388, 2019.

[37] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, pp. 131–154, 1994.

[38] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 308–333, 2017.

[39] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, 1995.

[40] C. Ji, Z. Chen, B. Xu, and Z. Zhao, "A novel method of mutation clustering based on domain analysis." in *SEKE*, 2009, pp. 422–425.

[41] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proc. ICSE*, 1993, pp. 100–107.

[42] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proc. ICSE*, 2010, pp. 435–444.

[43] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *Proc. ISSTA*, 2013, pp. 224–234.

[44] M. Jimenez, T. T. Checkam, M. Cordy, M. Papadakis, M. Kintis, Y. L. Traon, and M. Harman, "Are mutants really natural? a study on how" naturalness" helps mutant selection," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.

[45] J. M. Zhang, L. Zhang, D. Hao, L. Zhang, and M. Harman, "An empirical comparison of mutant selection assessment metrics," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2019, pp. 90–101.

[46] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: Better together," in *Proc. ASE*, 2013, pp. 92–102.

[47] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, no. 1, pp. 434–487, 2020.

[48] W. Ma, T. Laurent, M. Ojdanić, T. T. Chekam, A. Ventresque, and M. Papadakis, "Commit-aware mutation testing," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 394–405.

[49] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proc. ISSTA*, 2013, pp. 235–245.

[50] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *Proc. ISSTA*, 2012, pp. 331–341.

[51] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898–918, 2018.

[52] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *ISSTA*, 2016, pp. 342–353.

[53] C.-a. Sun, F. Xue, H. Liu, and X. Zhang, "A path-aware approach to mutant reduction in mutation testing," *Information and Software Technology*, vol. 81, pp. 65–81, 2017.

[54] C.-a. Sun, A. Fu, X. Guo, and T. Y. Chen, "Remusse: A redundant mutant identification technique based on selective symbolic execution," *IEEE Transactions on Reliability*, 2020.

[55] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proc. ISSTA*, 1993, pp. 139–148.

[56] A. Offutt VI and K. N. King, "A fortran 77 interpreter for mutation analysis," in *ACM SIGPLAN Notices*, vol. 22, no. 7. ACM, 1987, pp. 177–188.

[57] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *ISSTA*, 2014, pp. 315–326.

[58] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[59] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo, "Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation," in *ASE*, 2014, pp. 397–408.

[60] C.-P. Wong, J. Meinicke, and C. Kästner, "Beyond testing configurable systems: applying variational execution to automatic program repair and higher order mutation testing," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 749–753.

[61] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *ISSRE*, 2012, pp. 11–20.

[62] C.-P. Wong, J. Meinicke, L. Lazarek, and C. Kästner, "Faster variational execution with transparent bytecode transformation," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[63] C. H. P. Kim, S. Khurshid, and D. Batory, "Shared execution for efficiently testing product lines," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 221–230.

[64] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake, "On essential configuration complexity: Measuring interactions in highly-configurable systems," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 483–494.

[65] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel, "Multi-valued symbolic model-checking," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 4, pp. 371–408, 2003.

[66] M. d'Amorim, S. Lauterburg, and D. Marinov, "Delta execution for efficient state-space exploration of object-oriented programs," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 597–613, 2008.

[67] T. Kuchta, H. Palikareva, and C. Cadar, "Shadow symbolic execution for testing software patches," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 3, pp. 1–32, 2018.

[68] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 842–853.