



## Article

# EtWExplorer: Multi-Priority Scheduling Path Exploration Technology Based on Abstract Syntax Tree Analysis

Xinglu He , Pengfei Wang \*, Kai Lu and Xu Zhou 

College of Computer, National University of Defense Technology, Changsha 410073, China

\* Correspondence: pfwang@nudt.edu.cn

**Featured Application:** A multi-priority scheduling path exploration technology based on abstract syntax tree analysis.

**Abstract:** Symbolic execution is well known as a dynamic vulnerability discovery technique. Its greatest advantage is the capability to analyze the execution information of the program and to explore the path in the program deterministically. This is a more accurate way to determine if there are vulnerabilities in a program than randomized testing by fuzzing. In addition, symbolic execution does not suffer from the problem of decreasing the capability to discover new paths as more paths are discovered, similar to that caused by random-based fuzzing. However, the reason why symbolic execution is not widely used in vulnerability discovery is mainly due to the state space explosion in the program. The state space explosion severely affects the applicability of symbolic execution. To further improve the applicability of symbolic execution, this paper proposes a path exploration technology based on abstract syntax tree analysis. With the distance between the expression generated by the symbolic execution of the repeat location and the “unsatisfiable” condition of the “unsat” state, we can perform multi-priority scheduling for the repeat location state, thus mitigating the impact of the state space explosion on path exploration. We proposed and implemented EtWExplorer, a multi-priority scheduling technique based on abstract syntax tree analysis. With this technique, we can significantly improve the capability of symbolic execution to discover unknown paths even in state space exploration. Experiments show that EtWExplorer introduces a performance overhead of 72% in the worst case and can improve performance by 294% in the best case. EtWExplorer has a 95% improvement in state space explosion mitigation capability and a 199% to 983% improvement in the path exploration capability of block coverage and a 181% to 1047% improvement in the path exploration capability of edge coverage when facing programs that cause a state space explosion.



**Citation:** He, X.; Wang, P.; Lu, K.; Zhou, X. EtWExplorer: Multi-Priority Scheduling Path Exploration Technology Based on Abstract Syntax Tree Analysis. *Appl. Sci.* **2022**, *12*, 10182. <https://doi.org/10.3390/app121910182>

Academic Editor: Eui-Nam Huh

Received: 5 September 2022

Accepted: 8 October 2022

Published: 10 October 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

**Keywords:** vulnerability discovery; symbolic execution; state space explosion; abstract syntax tree analysis; multi-priority scheduling

## 1. Introduction

The symbolic execution technique can effectively find vulnerabilities in a program by testing each state of the program. Due to this advantage, symbolic execution techniques are widely used in competitions such as “Capture The Flag” (CTF) to analyze the execution state of the target program and try to discover vulnerabilities in it. Although the basic blocks of a program are finite during static analysis, the state space in most programs during dynamic execution is infinite. This leads to that symbolic execution is often used in small, simpler test programs, such as CTF. In the case of complex programs that people actually use in their daily lives, such infinite states will make it difficult for symbolic execution to discover potential vulnerabilities in the program.

At the present stage, symbolic execution appears mainly as an assistant tool in vulnerability discovery. It is widely used in hybrid-fuzzing as a supplement to fuzzing. Driller [1] provides a simple hybrid fuzzing idea. It introduces symbolic execution as an assist to

improve the coverage of vulnerability discovery when fuzzing is difficult to improve its coverage. QSYM [2] focuses on making symbolic execution more suitable for hybrid fuzzing usage cases. SAVIOR [3] proposes a bug-driven hybrid fuzzing approach. It focuses more attention on the parts of the code that have bugs by the undefined behavior sanitizer. HFL [4] improves the applicability of hybrid fuzzing and introduces hybrid fuzzing into kernel vulnerability discovery.

Symbolic execution could have provided more contributions to the vulnerability discovery. However, infinite states seriously limit the applicability of symbolic execution. Therefore, this paper proposes a multi-priority path exploration strategy based on abstract syntax tree (AST) analysis to mitigate the impact of these infinite states on the applicability of symbolic execution techniques.

We designed and implemented EtWExplorer, a multi-priority scheduling path exploration technique based on abstract syntax tree analysis. First, we identified the symbolic expressions generated in the program and recorded them. Then, we identified the constraints that could not be satisfied in the program and formed a correspondence between these unsatisfiable constraints and the identified symbolic expressions. When a basic block was repeatedly analyzed, we determined whether the symbolic expressions generated by the current basic block were related to the unsatisfiable constraints. Moreover, according to the corresponding analysis results, the basic blocks were classified into different priorities (this allows more meaningful program states to be executed with higher priority, while less meaningful program states will have lower priority). Thus, the impact of the state space explosion on symbolic execution was mitigated. Then, we can more easily explore the unknown paths in the program.

In summary, this paper contributes the following:

We analyzed that the cause of the state space explosion during symbolic execution was the massive growth of the state space caused by branching during repeated code execution. How to determine whether repeated code should be symbolically executed is the key problem to solving the state space explosion.

We propose an abstract syntax tree analysis approach that analyzes the positive direction of specific expressions in unsatisfied constraints. With the obtained positive direction, we can effectively classify the states generated in the symbolic execution with multiple priorities to mitigate the state space explosion in path exploration.

We designed a new path exploration technique. It analyzes the symbolic expressions generated in the program to prioritize the states generated in the symbolic execution. Ultimately, this prioritization is used to enhance the path exploration capability of symbolic execution.

We implemented the prototype tool EtWExplorer for the proposed multi-priority scheduling algorithm based on the abstract syntax tree analysis. Experiments show that EtWExplorer introduces a performance overhead of 72% in the worst case and can improve performance by 294% in the best case. EtWExplorer has a 95% improvement in the state space explosion mitigation capability and a 199% to 983% improvement in the path exploration capability of block coverage and a 181% to 1047% improvement in the path exploration capability of edge coverage when facing programs that cause a state space explosion.

## 2. Background

As the scale of the program continues to grow, symbolic execution is relatively difficult to use as a single tool for vulnerability discovery. However, symbolic execution has been increasingly used in hybrid fuzzing as an assistant tool to fuzzing. The path exploration capability of symbolic execution has been widely used. PANGOLIN [5] can assist fuzzing in generating test cases that satisfy the constraints and ensure that these generated test cases are evenly distributed in the solution space of the constraints. By using the information obtained from the symbolic execution phase, the ability of symbolic execution-assisted hybrid-fuzzing can be effectively improved. Directed symbolic execution [6] is also often used in vulnerability discovery. This also makes it more meaningful to improve the path exploration capability of symbolic execution.

### 2.1. State Space Explosion

State space explosion is common in symbolic executions. Symbolic execution forks the execution state into two instances each time a branch is encountered, one instance executing the true branch and the other instance executing the false branch. In general, this does not cause the state space explosion. However, if this branch is combined with a repeated code, symbolic execution forks a large number of instances of program execution state at local locations in the repeated code, which leads to an explosion of the program's state space. We illustrate this issue with the example in Listing 1.

**Listing 1.** State space explosion.

```

1 int main(int argc, char *argv[])
2 {
3     unsigned int i = (unsigned int)argv[1];
4     if (i > 10) return 0;
5     while (i < 15){
6         do_something();
7         i++;
8     }
9     return 0;
10 }
```

This program takes a value “i” from the input that can be controlled by the symbolic execution engine. When “i” is greater than 10, the program exits. When “i” is not greater than 10, the repeated code performs 5 to 15 times and generates 6 to 16 states, depending on the value of “i”. This is a simple state space explosion.

The number of states does not seem to be very large this is mainly due to the fact that the exit condition of the repeated code is 15. When the exit condition of the repeated code is 1010, thousands of states are generated. When these newly generated states can be terminated quickly (return 0) as in the example Listing 1, these generated states are not often subject to state space explosion. This situation is often found in programs designed based on procedure-oriented programming ideas. However, when these newly generated states are different from the example Listing 1 and difficult to terminate quickly, the state space of the program increases exponentially every time a new branch is encountered. This situation is often found in programs designed based on object-oriented programming ideas. The number of these growing states is difficult to estimate, thus causing a state space explosion. A tiny test program can trigger such a serious state space explosion. Moreover, in a real program, the problem will be even more serious.

### 2.2. Path Exploration Technique

Existing path exploration techniques are often a component of symbolic execution tools. On the one hand, more research has focused on improving the performance of symbolic execution tools and neglected the study of path exploration. The mainstream research directions of symbolic execution at this stage will be detailed in Section 7. On the other hand, the change in the positioning of symbolic execution in the vulnerability discovery domain has led to the fact that symbolic execution is no longer used as a single tool to perform vulnerability discovery. It often appears in the role of a fuzzing assistant tool in vulnerability discovery. As a result, the concolic execution [7], which focuses the analysis target on the concrete input (generated in the fuzzing phase) and does not generate a state space explosion, has become the dominant research direction for symbolic execution. This has led to a serious shortage of current research in the area of path exploration. However, the concolic execution can only fetch program paths around the path to which the concrete input is executed, which essentially limits the path exploration capability of the symbolic execution. The shortage of symbolic execution due to its ability to face the state space explosion has led to a change in its application positioning and the change in application positioning has led to a little study of path exploration techniques for symbolic execution at this stage.

At this stage, a part of the commonly used path exploration techniques originates from the traditional tree search algorithm. This is due to the fact that the execution of the program is a process of searching from the root node (entry point) to the leaf node (exit point) of the binary tree. Each branch of the program is a node of the binary tree (except for the leaf nodes). Therefore, the traditional tree search algorithm is the most common type of path exploration technique.

Another class of path exploration techniques oriented to symbolic execution often require the assistance of control flow graphs (CFG) to explore the paths in a program effectively. Angr [8] proposes a path exploration technique that enhances the path exploration ability with the help of control flow graphs generated by Angr. Veritesting [9] avoids state space explosion by constructing a local CFG, finding a specific node in the CFG, and directing the symbolic execution to the target location. These two path exploration techniques allow for more efficient discovery of paths in programs based on CFGs. However, both techniques require precise CFGs. Even the emulated CFG implemented in Angr is not sufficient to support its exploration technique. Veritesting generates a local CFG and is not able to analyze the CFG accurately in the face of some unknown nodes and system calls. Therefore, such CFG-based path exploration techniques are difficult to really solve the state space explosion problem in real-world programs.

There is also a class of path exploration techniques that require manual assistance to enhance the path exploration capabilities of symbolic execution. Manual merge point technology is one of these. It is done by manually marking an address in the program as a merge point, so states that reach that address will be briefly held, and any other states that reach that same point before timeout will be merged. This requires researchers to have an extremely deep knowledge of the program in order to set the merge point accurately. This deep knowledge is difficult to achieve. Even if the merge point is set accurately, the excessive merging of programs during dynamic execution is likely to lead to a loss of symbolic execution path exploration capability.

Therefore, this paper proposes a multi-priority scheduling method based on abstract syntax tree analysis to mitigate the state space explosion problem in path exploration of symbolic execution.

### 3. Design

In this section, we illustrate how to implement a multi-priority scheduling algorithm based on abstract syntax tree analysis to mitigate the state space explosion problem encountered in path exploration techniques.

Before illustrating the technical details, we will first discuss how to define the coverage of symbolic execution. Most programs have an immeasurable number of paths. Therefore, using path coverage as a coverage metric cannot really separate the advantages and disadvantages of exploration techniques. In this paper, we use block coverage and edge coverage, which are commonly used in the fuzzing area, as metrics for path exploration techniques. The exploration techniques that can find more basic blocks and more edges are good techniques.

#### 3.1. The Difficulty of State Space Explosion Mitigation

In Section 2, we briefly discuss the causes of state space explosions. When a symbolic execution reaches repeated code, it forks itself at the branch in the repeated code, and the more times the repeated code is executed, the more the program execution state is forked. This repeated code will cause a state space explosion.

One of the most straightforward ideas to mitigate the state space explosion is to symbolic execute the repeated code only a minimum number of times when the symbolic execution enters the repeated code. The state space explosion of the symbolic execution is mitigated by reducing the state generated in the repeated code by exiting it as soon as possible. The core idea of this approach is that the state space of a program may be infinite; however, the number of basic blocks must be finite. If the states caused by repeated

code are eliminated, then the states generated by a finite number of basic blocks must also be finite. Thus, it will not lead to a state space explosion. However, this approach will severely reduce the path exploration capability of the symbolic execution. This reduction in path exploration capability will affect the performance of path exploration more severely than the state space explosion.

We made some modifications to the example used in Section 2. In Listing 2, we set “function one” to be executed when “j” is greater than 10. If our symbolic execution tool only performs the minimum repeated code exploration. When the initial value of “i” is set to 10, the repeated code will only be executed 5 times. The maximum value of “j” is 5, which does not satisfy the constraint that “j” is greater than 10 for the execution of “function one”. This causes the symbolic execution to fail to detect the state of “function one”, resulting in a reduction in path exploration capability.

**Listing 2.** State space explosion with a reachable edge.

```

1 int main(int argc, char *argv[])
2 {
3     unsigned int i = (unsigned int)argv[1];
4     unsigned int j = 0;
5     if (i > 10) return 0;
6     while (i < 15){
7         do_something();
8         if (j > 10) function_one();
9         i++;
10        j++;
11    }
12    return 0;
13 }
```

We could propose a new idea on this base. We determine whether it is necessary to explore the repeated code by identifying whether there are any unexecuted edges following the repeated code. In this way, the repeated code can be explored continuously while “function one” is not executed, and further exploration of the repeated code can be stopped when “function one” is executed. By exploring the repeated code only a meaningful (new paths can be found) number of times, the number of state spaces is reduced while ensuring accuracy. However, this idea is still not effective in mitigating the state space explosion when used in practice.

We have made some more modifications to the example we used in Listing 2. In Listing 3, we set “function two” to be executed when “k” is greater than 20. Therefore, “k” cannot be greater than 20. There is an edge following the repeated code that can never be reached. This makes the exploration of symbolic execution to try to find the “function two” meaningless and hard to terminate. Symbolic execution tools still encounter the problem of exploding state spaces.

Therefore, we need a more precise approach that can effectively reduce the state space explosion of the symbolic execution without losing its path exploration capability. We propose a technique based on abstract syntax tree analysis that allows classifying the program execution states into different priorities. By classifying the states with different priorities, the symbolic execution can be more focused on the states that are more likely to discover new paths. Moreover, low-priority states will be difficult to be executed or even delete, thus avoiding state space explosion and saving memory space.

Through the above two examples, we can find that the biggest difficulty in mitigating the path explosion problem is that too much analysis of repeated code will lead to the state space explosion; too little analysis of repeated code will weaken the accuracy of path exploration. Therefore, proposing a technique that can effectively determine whether to further symbolic execution of repeated code is the key problem to mitigating the state space explosion. Using the unexecuted edges following the repeated code as the discriminant of whether to perform symbolic execution on the repeated code cannot satisfy the complex

situations in real-world programs. Therefore, we propose a discriminative approach based on the abstract syntax tree analysis to solve this problem.

**Listing 3.** State space explosion with an unreachable edge.

```

1 int main(int argc, char *argv[])
2 {
3     unsigned int i = (unsigned int)argv[1];
4     unsigned int j = 0;
5     unsigned int k = 0;
6     if (i > 10) return 0;
7     while (i < 15){
8         do_something();
9         if (j > 10) function_one();
10        if (k > 20) function_two();
11        i++;
12        j++;
13        k++;
14    }
15    return 0;
16 }

```

### 3.2. Abstract Syntax Tree Analysis

From the above description, we know that the key to solving the state space explosion is to accurately determine whether to continue the execution and analysis of a code that has already been executed at least one time. The more the code is repeatedly executed, the more likely it is that the state space of symbolic execution will explode; the less the code is repeatedly executed, the more likely it is that the accuracy of symbolic execution will decrease. Therefore, we need an approach that can assist us in deciding whether the repeated code should be executed again, to achieve a balance between the number of state spaces and exploration accuracy. To solve this problem, we propose the abstract syntax tree analysis approach to mitigate the state space explosion problem in symbolic execution.

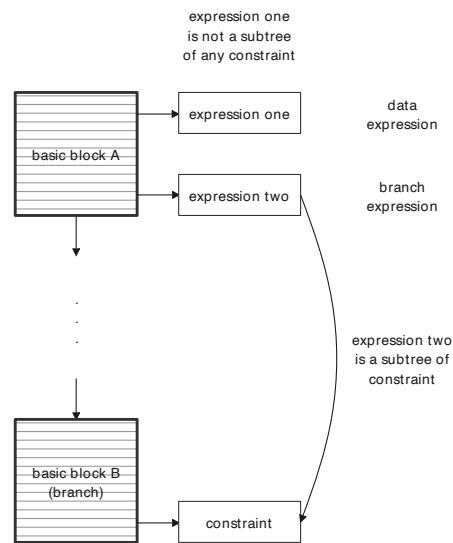
#### 3.2.1. Identify Critical Expressions

Whenever a basic block is executed, a large number of expressions are generated. We divide these generated expressions into two categories: data expressions and branch expressions.

As shown in Figure 1, data expressions are expressions that perform operations on data and calculate the result of program execution; branch expressions are expressions that control the branches during the program execution. One of the easiest ways to identify whether an expression is a data expression or a branch expression is to check whether it is a subtree of a branch constraint. Data expressions do not cause the state space explosion, and branch expressions combined with repeated code are the root cause of the state space explosion.

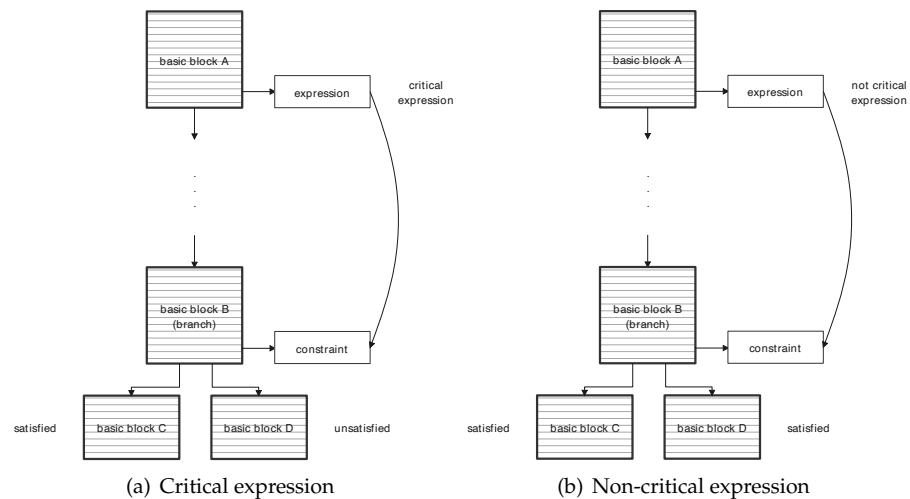
Some expressions may be both data expressions and branch expressions. Therefore, it is difficult to accurately determine whether an expression is a data expression or a branch expression when the expression is just generated. The other problem is that some branch expressions affect loose branch constraints and are already satisfied without further execution. For the basic block that generates such an expression, there is no need to repeat the execution and analysis.

Our ultimate goal is to improve the coverage of path exploration techniques, and paths that were previously unexecutable are the focus of our interest. Therefore, the essential reason for whether we repeat execution and analysis for a basic block is whether a basic block generates a branch expression that affects the unsatisfiable constraints and arrives at an unexecuted path. We need to identify these branch expressions in the program.



**Figure 1.** Data expressions and Branch expressions.

As shown in Figure 2, we call such branch expressions in Figure 2a *critical expressions*. The most important feature of the critical expression is that one of the branch paths that it affects is unsatisfied (in general, there is no branch constraint that both paths are unsatisfied). This is because unsatisfiable constraints represent undiscovered paths. We consider that the operation of expressions that affect these unsatisfiable constraints has the potential to turn unsatisfiable constraints into satisfiable constraints and thus discover new paths. Therefore, we consider such expressions as critical expressions. Moreover, the branch expressions in Figure 2b are non-critical. Both paths of the branch nodes it affects are already satisfied and no new basic blocks and edges can be found. Such expressions are not interesting to us. Therefore, it is not a critical expression.



**Figure 2.** Identify critical expressions.

### 3.2.2. Prioritization of Symbolic Execution States

In this section, we have an assumption that when the basic block is repeatedly executed, the first time its expressions are computed as part of the constraint, its expressions are also computed by the same constraint during the repeated execution. Therefore, not all basic blocks containing critical expressions are worthy of repeated symbolic execution and analysis. As the simplest example, if the generated expressions are exactly the same in the two symbolic executions before and after, then the repeated executions are meaningless. So, after identifying the critical expressions, we also need to analyze the expressions generated by

the two executions before and after to determine whether the repeated symbolic executions are meaningful.

We show the main idea of our multi-priority scheduling algorithm in Algorithm 1. We illustrate our approach to determine whether the symbolic execution of repeated basic blocks is meaningful.

---

**Algorithm 1:** Multi-priority scheduling algorithm

---

**Input:** Current execution state  $State_{current}$ , Previously executed basic block information  $I_{previous}$  (Dictionary)  
**Result:** Successor execution states  $NewStates$  (List), priority of successor execution states  $Priority$ , newly executed basic block information  $I_{new}$  (Dictionary)

```

1  $NewStates, expr \leftarrow SymbolicExecution(State_{current});$ 
2 if  $GetStateAddr(State_{current})$  in  $GetDictKeys(I_{previous})$  then
3    $PreviousStateInfo \leftarrow GetInfoByAddr(I_{previous}, GetStateAddr(State_{current}));$ 
4   if  $StateInfoIsCritical(PreviousStateInfo)$  then
5      $PositiveDirection \leftarrow GetPositiveDirection(PreviousStateInfo);$ 
6     if  $PositiveDirection == 'not\ sure'$  then
7        $Priority \leftarrow 'Middle';$ 
8     else if  $PositiveDirection == '+'$  then
9       if  $expr > All(GetExprsByInfo(PreviousStateInfo))$  then
10         $Priority \leftarrow SetPriority(NewStates, 'Positive');$ 
11      else if  $expr < All(GetExprsByInfo(PreviousStateInfo))$  then
12         $Priority \leftarrow SetPriority(NewStates, 'Negative');$ 
13      else
14         $Priority \leftarrow SetPriority(NewStates, 'Middle');$ 
15      end
16    else if  $PositiveDirection == '-'$  then
17      if  $expr < All(GetExprsByInfo(PreviousStateInfo))$  then
18         $Priority \leftarrow SetPriority(NewStates, 'Positive');$ 
19      else if  $expr > All(GetExprsByInfo(PreviousStateInfo))$  then
20         $Priority \leftarrow SetPriority(NewStates, 'Negative');$ 
21      else
22         $Priority \leftarrow SetPriority(NewStates, 'Middle');$ 
23      end
24    end
25  else
26     $Priority \leftarrow SetPriority(NewStates, 'Non-critical');$ 
27  end
28 else
29    $I_{new} \leftarrow SetInfoByAddr(I_{previous}, GetStateAddr(State_{current}), expr);$ 
30    $Priority \leftarrow SetPriority(NewStates, 'Not\ executed');$ 
31 end
32 for SuccessorState in  $NewStates$  do
33   if  $StateSatisfiable(SuccessorState) == False$  then
34      $UnsatConstraint \leftarrow GetUnsatConstraintFromState(SuccessorState);$ 
35     for StateAddr, Exprs in  $GetKeyValueFromDict(I_{previous})$  do
36       if  $Any(Exprs)$  in  $GetChildrenAST(UnsatConstraint)$  then
37          $I_{new} \leftarrow MarkStateInfoAsCritical(I_{previous}, StateAddr, Exprs);$ 
38       end
39     end
40   end
41 end

```

---

The first task of the multi-priority scheduling algorithm is to prioritize the states for symbolic execution. After a state is symbolically executed, we first determine if the basic block where the state is located is being executed for the first time in line 2. If this is the first time the basic block is executed, we record the address and the expression generated during the execution into the  $I_{new}$  and set the priority as “Not executed” in lines 29 and 30. If not, we will determine if the expression generated at the previous same position is critical in lines 3 and 4. After that, we calculate the positive direction of the current position based on the previous state information of the same position in line 5. We decide the priority of the state based on whether the relationship between the newly generated expression and the previously generated expression matches the positive direction in lines 6 to 24. The result of the priority affects the current state’s successor state. We use this priority to mitigate the impact of state space explosion on path exploration capability.

Another major task of the multi-priority scheduling algorithm is to identify critical expressions. We will determine if there is an unsatisfiable state in the new state generated from the current state in line 33. If it exists, we obtain the constraint that causes the state to be unsatisfiable in line 34. Then, we search in the  $I_{previous}$  if any expression is a subtree of this unsatisfiable constraint in lines 35 and 36. If such an expression exists, the address of the basic block in which this expression is generated and the expression are marked as critical expression information and recorded in  $I_{new}$ .

With this approach, we set up five priorities to mitigate the impact of state space explosion on symbolic execution based on the impact of critical expressions on unsatisfiable constraints.

- Not executed: The basic block is executed for the first time;
- Have positive expressions: The modification of the expression brings the unsatisfiable constraint closer to being satisfied (For example, the first time execution of the initial constraint “ $i \leq 10$ ”, the basic block generates the expression “ $i + 1$ ”, which does not satisfy constraint “ $i + 1 \geq 15$ ”; the second time execution generates the expression “ $i + 2$ ”, which does not satisfy constraint “ $i + 2 \geq 15$ ”. Since “ $i + 2$ ” is closer to the constraint not less than 15 than “ $i + 1$ ”, we consider this repeated execution as positive.);
- Have negative expressions: The modification of the expression brings the unsatisfiable constraint farther from being satisfied;
- Have no critical expressions: The basic block does not generate expressions or the generated expressions are not related to unsatisfiable constraints;
- Have middle expressions: The distance of the expression generated by the basic block from the unsatisfiable constraint is difficult to define, or the generated expression is neither closer to nor further from the unsatisfiable constraint than previous expressions.

The multi-priority scheduling can effectively help us mitigate the problem of state space explosion during symbolic execution. First, we consider the basic blocks that have not been symbolically executed to have the highest priority. Exploring new paths is our ultimate goal. Second, we consider the basic blocks containing positive critical expressions have higher priority. Positive critical expressions are more helpful for us to satisfy previously unsatisfiable constraints. Third, we put the priority of the basic blocks containing negative critical expressions in the middle. The total number of negative basic blocks is generally small, it is difficult to trigger the state space explosion. At the same time, some errors in the design implementation of abstract syntax tree analysis can be avoided.

Then, we put the priority of the non-critical expression basic block before the priority of the middle expression basic block. This is due to the fact that many positive critical expressions are not used in constraints immediately after they are generated. There may be multiple non-critical expression basic blocks between the generation and use of positive critical expressions. Therefore, we believe that not generating critical expressions is preferred over generating middle critical expressions. At the same time, to ensure the effectiveness of non-critical expressions, we keep only the successor states of the latest positive critical

expressions basic block. Moreover, the old non-critical expression basic block states will be deleted.

This approach of lowering the priority and deleting can significantly reduce the generation of state space of the program, thus mitigating the impact of state space explosion on path exploration.

### 3.2.3. Analyze Abstract Syntax Trees for Positive Direction

We adopt a simple heuristic to determine the positive direction of an expression in an unsatisfiable constraint.

When analyzing the abstract syntax tree, we first analyze the conditions of unsatisfiable expressions. If the condition of the unsatisfiable constraint is greater than or not less than, it means that the constraint cannot be satisfied because the side where the expression is located is less than a specific value, and the direction of increase on the side where the expression is located is positive. If the condition of the unsatisfiable constraint is less than or not greater than, it means that the constraint cannot be satisfied because the side where the expression is located is greater than a specific value, and the direction of decrease on the side where the expression is located is positive. If the condition of the unsatisfiable constraint is equal, we symbolically compare the size relationship between the two sides of the condition in the constraint and use this to determine the positive direction of the side on which the expression is located. If other unknown conditions are encountered, we return an indeterminate identifier, indicating that the positive direction of the expression cannot be analyzed accurately.

When we obtain the positive direction of the side where the expression is located, we need to determine the positive direction of the expression further. We iteratively analyze the abstract syntax tree on the side where the expression is located and step-by-step analyze the sign (positive or negative) on the side where the expression is located.

Eventually, the number of negative signs combined with the positive direction (on the side where the expression is located) is used to determine the positive direction of the expression in the unsatisfiable constraint.

### 3.2.4. Further Satisfiability Analysis

After obtaining the positive direction of the expression, we will perform a further satisfiability analysis of the constraints. First, when there are multiple same expressions for an unsatisfiable constraint, we require that the expressions are on the same side of the unsatisfiable constraint. If there are the same expressions on both sides of the unsatisfiable constraint, then modifying the expressions at the same time will not affect the satisfiability of the constraint. After that, we also compare the satisfied constraint with the unsatisfiable constraint in the “unsat” state. If the two conditions are equal on the side, which contains the expression, we consider that the constraint in this state is hard to unsatisfiable. When these two cases occur, we will unmark the critical expression flag for the relevant expressions. With these two satisfiability analyses, we can effectively cull a large number of unsatisfiable constraints in the program, e.g.,  $i < 15$  AND  $i > 20$  in the third example code. Finally, when we have found a satisfiable path to the previous “unsat” state, we will also unmark the critical expression flag for the relevant expressions. With these approaches, we can further improve the accuracy of multi-priority scheduling.

## 3.3. How to Work

The pure algorithm used to illustrate how EtWExplorer works does not seem very clear. Therefore, we briefly describe the workflow of EtWExplorer, using Listing 4 as an example.

**Listing 4.** Path exploration example.

```

1 int main(int argc, char *argv[])
2 {
3     unsigned int loop_count = (unsigned int)argv[1];
4     if (loop_count >= 10) return 0;
5     while (1) {
6         loop_count++;
7         if (loop_count >= 11) {
8             func1();
9             break;
10        }
11        if (loop_count >= 12) {
12            func2();
13            break;
14        }
15    }
16    return 0;
17 }

```

When the program reaches line 3, we know that “loop count” is a symbolic variable we can control, and a symbolic expression “symbolic one” is generated. When the program reaches line 4, we encounter a branch condition. Here, we are only concerned with the situation where “loop count” is less than 10. So, when the program reaches line 5, the state constraint is “symbolic one < 10”.

At this point, the program enters the loop for the first time. When the program reaches line 6, the expression in “loop count” is changed to “symbolic one + 1”. When the program reaches line 7, the program encounters a new branch condition, “loop count >= 11”. Since the expression in “loop count” is “symbolic one + 1”, we find that “symbolic one < 10” and “symbolic one + 1 > 11” is not satisfied. We identify the expression in “loop count” as the critical expression. At the same time, we analyze why the condition is unsatisfiable. We find that this is due to that “symbolic one + 1” is less than 11. Then, we generate a “+” as the positive direction of “loop count”. When the program reaches line 11, we perform a similar operation to the code in line 6.

Then, the program enters the loop for the second time. When the program reaches line 6 for the second time, we find that the expression in the critical expression “loop count” has changed from the expression in the first execution. The new expression “symbolic one + 2” is larger than the old expression “symbolic one + 1”. This is consistent with the positive direction of “loop count” from the previous analysis, which means that it is meaningful to continue the execution and repeat the analysis of the current basic block. When the program reaches line 7 for the second time, the expression of “loop count” is “symbolic one + 2”, we find that “symbolic one < 10” and “symbolic one + 2 > 11” are satisfied. The unsatisfiable constraint in line 7 is converted into satisfiable, and “loop count” is unmarked as the critical expression. Since “loop count” is still the critical expression in the unsatisfiable condition in line 11. Therefore, “loop count” is still a critical expression at this time. When the program reaches line 11 for the second time, we find that the constraints are “symbolic one < 10”, “symbolic one + 2 < 11”, and “symbolic one + 2 >= 12”. Further satisfiability analysis shows that “symbolic one + 2” cannot satisfy both less than 11 and greater than or equal to 12. Therefore, we set the constraint in line 11 to totally unsatisfiable, and “loop count” is unmarked as the critical expression. At this time, the “loop count” is not a critical expression.

When the program reaches line 6 for the third time, we find that the program does not operate on the critical expression. We consider such symbolic execution to be relatively meaningless and prune the state of such symbolic execution. At this point, if line 4 is not “return 0” or it is a function, the symbolic execution can focus more on the execution there.

With this approach, we complete the analysis in 2 loops for this code, which may require 10 loops to complete with traditional approaches. It dramatically reduces the number of states generated in symbolic execution, thus mitigating the impact of the state

space explosion on the symbolic execution path exploration capability. In real-world programs, the complexity of the program will be much higher, and EtWExplorer will become more effective in mitigating state space explosion.

#### 4. Implementation

Our symbolic execution engine is Angr and the satisfiability solver is the Z3 solver [10]. For better path exploration, we have made some changes to Angr. First, we modified Angr's memory operations to limit Angr's memory malloc capability to prevent the program from taking a lot of time when making memory malloc with symbolic size. Then, a new symbolic execution mode was created to facilitate better symbolic execution oriented to path exploration. After that, we added a callback function when Angr generates the "project" object, through which we can modify the alias of the function called in the dynamic link library. It is possible to make the functions in the program better match with the SimProcedures implemented by Angr. At last, we have modified Angr's file reading system. When a program needs to read a file that is not a user-assigned symbolic file, Angr will look for the file with the same path in the local machine and pass the file to the program if it exists, or return none or a symbolic file depending on the configuration if it does not exist. With these modifications, we have improved Angr's applicability to explore paths in real-world programs.

Besides the above modifications to Angr, we also propose a coverage-based path exploration technique since the ultimate goal we want to achieve is to improve the path coverage of symbolic execution. In this technique, undiscovered basic blocks and edges will be symbolically executed with more priority. This coverage-based path exploration technique is also presented as a control path exploration technique in the experiments.

Eventually, we implemented our multi-priority scheduling path exploration tool, EtWExplorer, based on Python.

#### 5. Evaluation

We run all our evaluations on a 64-bit server running Ubuntu 18.04.5 LTS with two Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz CPUs (16 cores in total) and 125 G memory. Our evaluation aims to answer the following questions.

1. **Performance overhead:** How much performance overhead does EtWExplorer introduce compared to traditional path exploration technology?
2. **States space explosion mitigation:** How much more effective does EtWExplorer mitigate state space explosion compared to traditional path exploration techniques?
3. **Path exploration capability improvement:** How much improvement is EtWExplorer's path exploration capability compared to traditional path exploration technology?

Since EtWExplorer needs to analyze and record the abstract syntax tree during execution. In this way, it reduces the priority of a part of the states and even prunes a part of meaningless states when the state space explodes. EtWExplorer is essentially a time-efficiency trade-off technology. Therefore, our experiments tested EtWExplorer in two areas: performance overhead and path exploration capability. For the control group, we select some common path exploration techniques.

1. **Lengthlimiter technique:** Length-limited breadth-first exploration technique. We use this technique as a representative of path exploration techniques based on tree search algorithms.
2. **Veritesting [9] technique:** The local control flow graph-based path exploration technique. We use this technique as a representative of path exploration techniques for generating local control flow graphs.
3. **Angr's Explorer [8] technique:** The global control flow graph-based path exploration technique. We use this technique as a representative of path exploration techniques for generating global control flow graphs.
4. **Coverage-based technique:** Coverage-based path exploration technique. We use this technique as a representative of path exploration techniques based on coverage.

Here, we did not use the manual merge point technique as a control group. This is due to the fact that the manual merge point technique requires manual assistance. We consider it fairer to experiment without manual assistance for all tested techniques.

We used some real-world programs as the test set for this paper. For different experimental purposes, we divide the test programs into two categories. One category involves programs that do not cause a state space explosion. We use this category of programs to test the performance overhead introduced by EtWExplorer. The other category involves programs that cause a state space explosion. We use this category of programs to test the priority scheduling and pruning strategy to improve path exploration capability. We choose “ffmpeg”, “ffprobe” in the ffmpeg and “nm”, “objcopy”, “objdump”, “readelf”, “size”, “strings” in the binutils to represent the first category of the programs; choose “guetzli”, “jhead” and “pdfdetach”, “pdfimages”, “pdftinfo”, “pdftotext” in the xpdf to represent the second category of the programs. By testing different categories of programs, we can better understand EtWExplorer’s effect on different categories of programs.

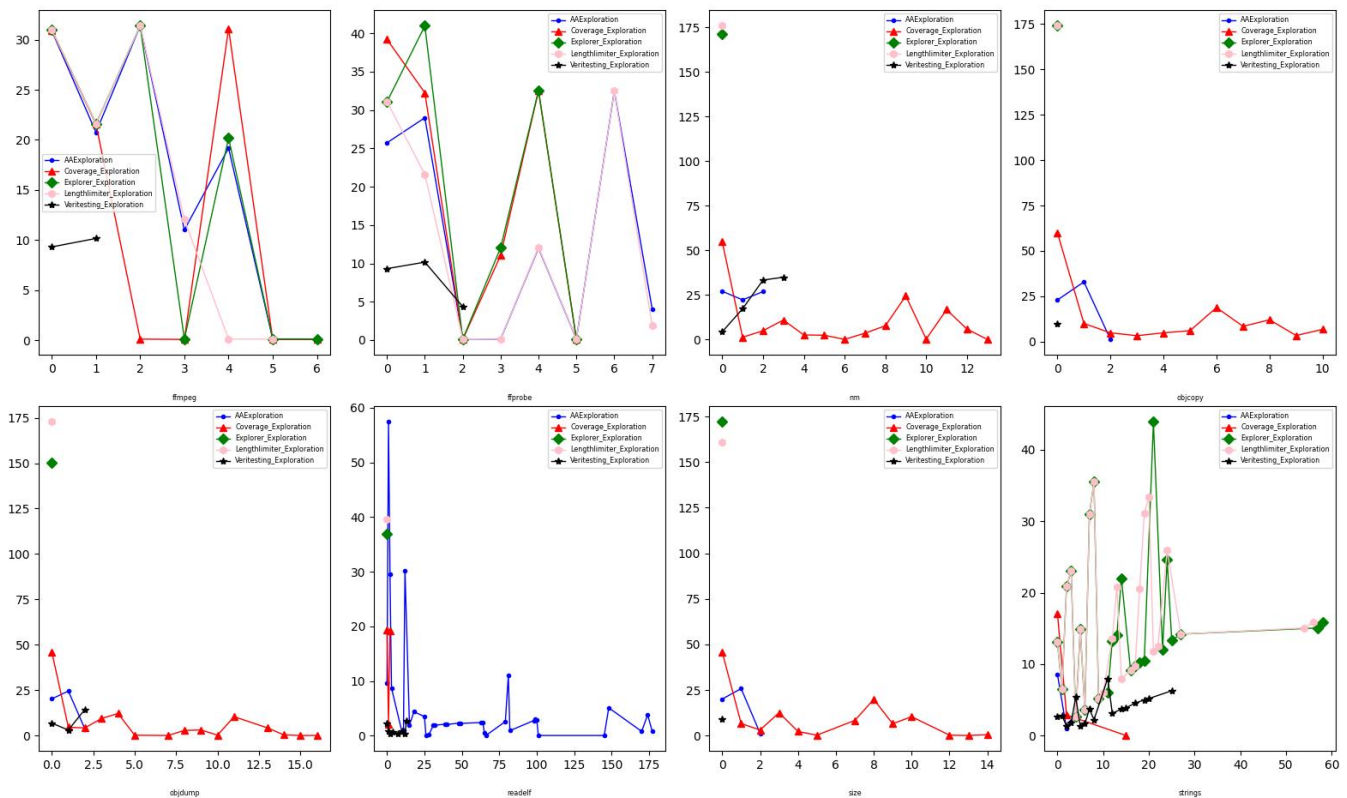
Since there are exploration approaches whose performances can be severely affected by state space explosion, we put a limit on the execution time and the amount of memory space used for the experiments. In this paper, the execution time of the path exploration capability test was limited to 6 h and the maximum memory space used was 20 GB. If the limit is over, the path exploration will be automatically terminated.

### 5.1. Performance Overhead

EtWExplorer is essentially a time-efficiency trade-off technology. In order to improve the path exploration capability of EtWExplorer, we introduce the abstract syntax tree analysis technique to assist us. The abstract syntax tree analysis technique is used to record and analyze the expressions generated by each executed basic block and match them with the unsatisfiable constraints of the “unsat” basic blocks. This affects the execution efficiency of the path exploration technique to some degree. Therefore, we illustrate how much performance overhead is introduced by EtWExplorer, which is based on abstract syntax tree analysis techniques, through this experiment.

In order to better test the performance overhead introduced by EtWExplorer, we needed to make some limitations on the test program. We set the test program that did not cause a state space explosion. This was due to the fact that, in this experiment, we were more concerned with the execution efficiency of the program rather than the path exploration capability. Without causing a state space explosion, the path exploration capabilities of several path exploration techniques were approximately the same, we only needed to compare the time length of path exploration to obtain the performance overhead introduced by EtWExplorer. Therefore, we chose “ffmpeg”, “ffprobe” in the ffmpeg and “nm”, “objcopy”, “objdump”, “readelf”, “size”, “strings” in the binutils as the test program.

In Figure 3, the x-axis represents the explore technology execution time in minutes, and the y-axis represents the number of states executed per second. The blue line represents the number of states executed per second of EtWExplorer, the red line bar represents the number of states executed per second of our coverage technique, the green line represents the number of states executed per second of Angr’s Explorer technique, the pink line represents the number of states executed per second of the Lengthlimiter technique, and the black line represents the number of states executed per second of the Veritesting technique. We will make two notes here. First, the end times of different exploration techniques are not necessarily the same because of their different execution efficiencies and methods of deciding the completion of execution. Second, in some test programs (especially “ffmpeg”, “ffprobe” in the ffmpeg), there are encode/decode operations, and symbolic executions of such concentrated data operations will seriously affect efficiency. This results in a situation where the execution is sometimes fast and sometimes slow during symbolic execution. Therefore, the path exploration performance of Veritesting performs weakly in the experiments.



**Figure 3.** Performance overhead when exploring programs that do not cause state space explosion.

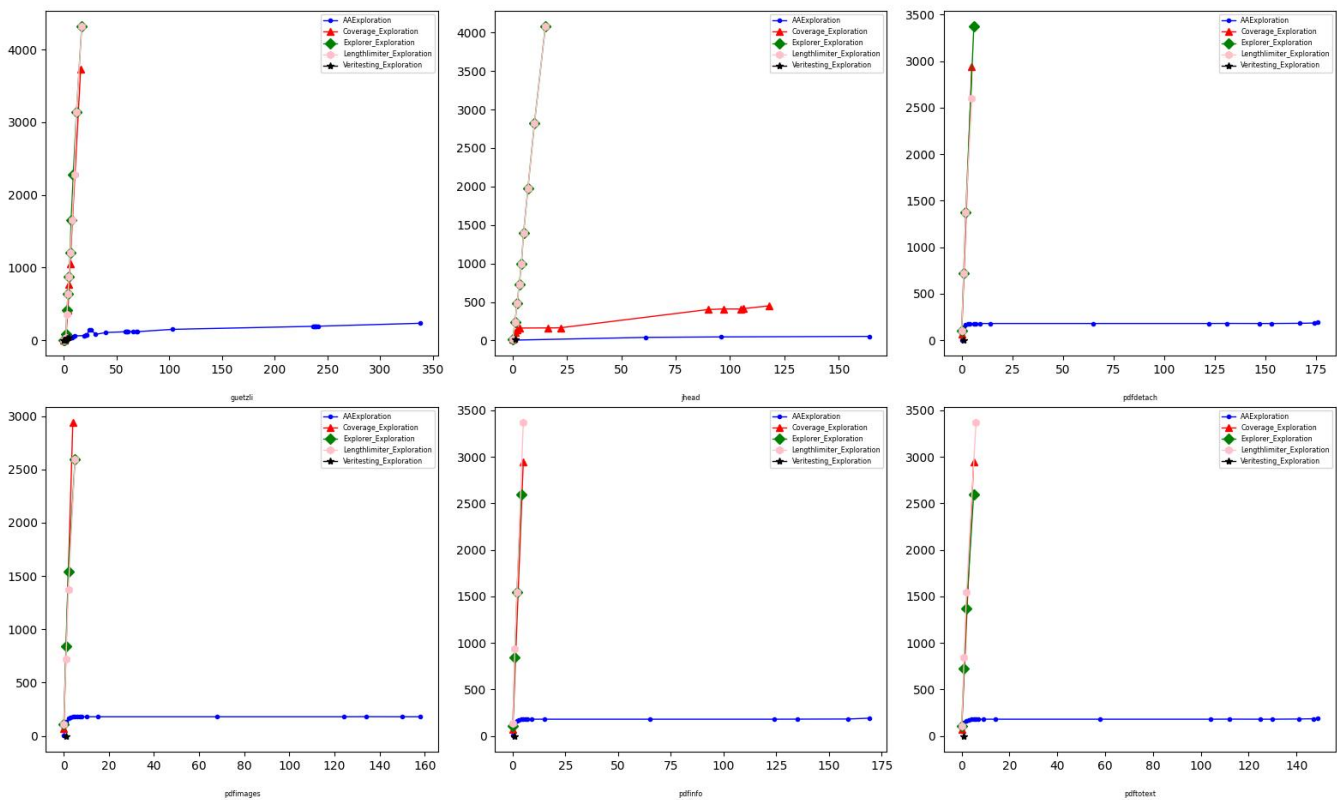
As we can see from the graph, EtWExplorer did not introduce significant performance overhead. In most programs, EtWExplorer’s exploration capability is similar to that of other techniques. We believe that this is mainly due to the fact that EtWExplorer does not introduce a lot of abstract syntax tree analysis when the program does not cause a state space explosion. Therefore, no significant performance overhead was introduced.

Finally, we calculated a mean value by adding up the results obtained from the eight tests. The execution efficiency (average number of state symbolic executions per second) ratio was the EtWExplorer technique:Coverage-based technique:Angr’s Explorer technique:Lengthlimiter technique:Veriteesting technique = 215:156:754:759:73. (Angr’s Explorer’s global control flow graphs are generated offline, while Veriteesting’s local control flow graphs were generated online, so there was a big difference in performance between the two techniques.) EtWExplorer introduced a performance overhead of 72% in the worst case and could improve performance by 294% in the best case. We can conclude that the performance overhead introduced by EtWExplorer was within a moderate and reasonable range.

## 5.2. States Space Explosion Mitigation

The capability of the proposed multi-priority scheduling technique for state space explosion mitigation is another test point of this paper. Therefore, in this part of the experiment, the programs we chose to test programs caused a state space explosion. Therefore, we chose “guetzli”, “jhead” and “pdfdetach”, “pdfimages”, “pdfinfo”, “pdftotext” in the xpdf as the program under test.

In Figure 4, the x-axis represents the symbolic execution time in minutes, and the y-axis represents the number of states stored in the symbolic execution engine. The meaning represented by the lines of each color in the figure is the same as in Figure 3.



**Figure 4.** State space explosion mitigation capability when exploring programs that cause state space explosions.

As we can see from the graph, many programs encounter the problem of state space explosion when performing the symbolic execution. Many path exploration techniques encounter the state space explosion problem when they just start to explore the execution path of a program. Then, their memory usage reaches the threshold we set and exits execution. To clarify here, Veriteesting requires the construction of local control flow graphs during the runtime. Moreover, precise local control flow graphs are difficult to construct. Therefore, in many cases, Veriteesting will not only encounter the problem of state space explosion of the program during execution but also the difficulty of control flow graph construction. As a result, the memory threshold is reached at the start of the program. Therefore, Veriteesting exits execution when the number of states is still small.

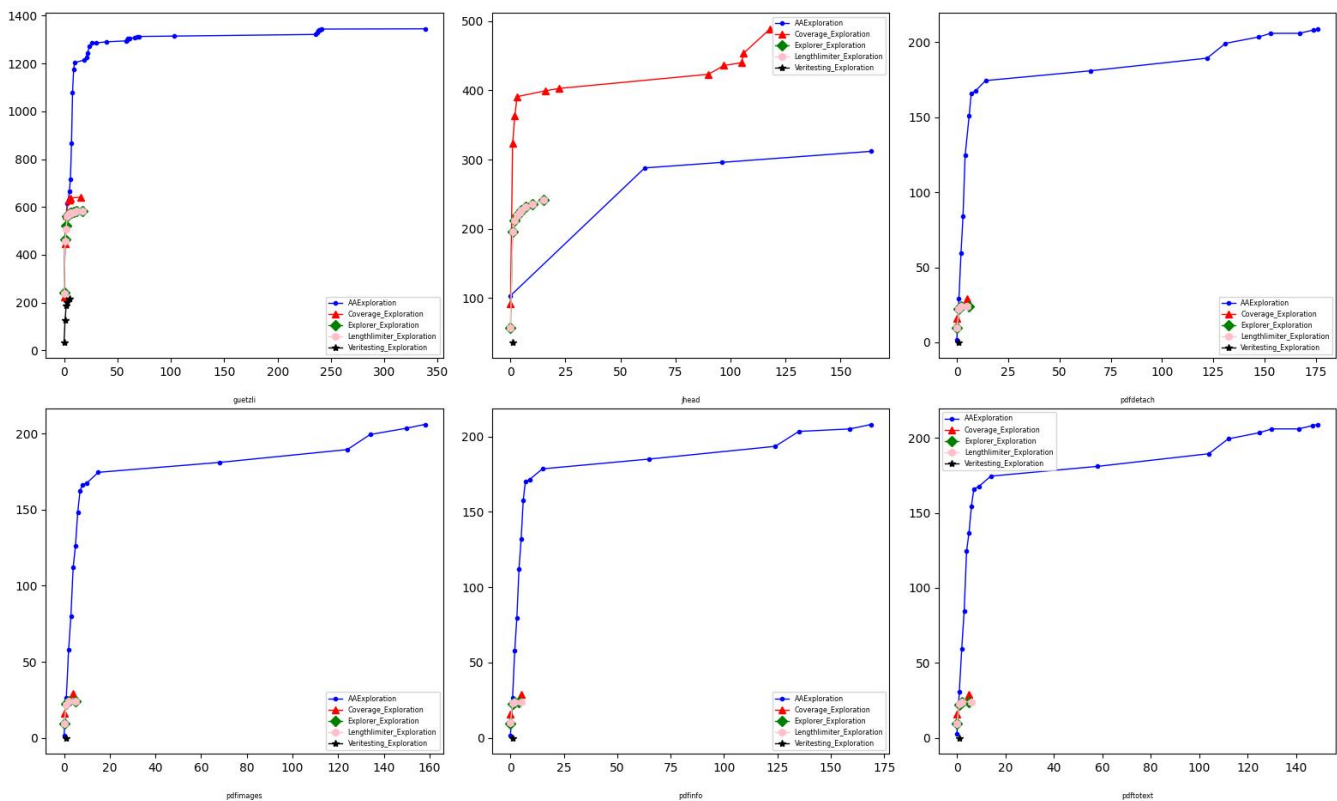
This is the most common situation in our symbolic execution, when the state space of the program expands dramatically, the computer's memory cannot satisfy it, and the symbolic execution cannot continue. In contrast, our proposed multi-priority scheduling strategy based on abstract syntax tree analysis can effectively mitigate the state space explosion problem in symbolic execution. Moreover, it can maintain the symbolic execution to continue.

Finally, we calculated a mean value by adding up the results obtained from the six procedures. The state space explosion mitigation capability ratio of path exploration is EtWExplorer technique:Coverage-based technique:Angr's Explorer technique:Lengthlimiter technique:Veriteesting technique = 1035:15,952:19,553:20,329:55. EtWExplorer's state space explosion mitigation capability improved by about 95%, excluding Veriteesting. We can clearly conclude that EtWExplorer can effectively mitigate the state space explosion problem in symbolic execution. This is even if we limit memory and the control group techniques exit at the beginning of execution. In a memory unlimited environment, EtWExplorer's state space explosion mitigation capabilities would be exponentially improved.

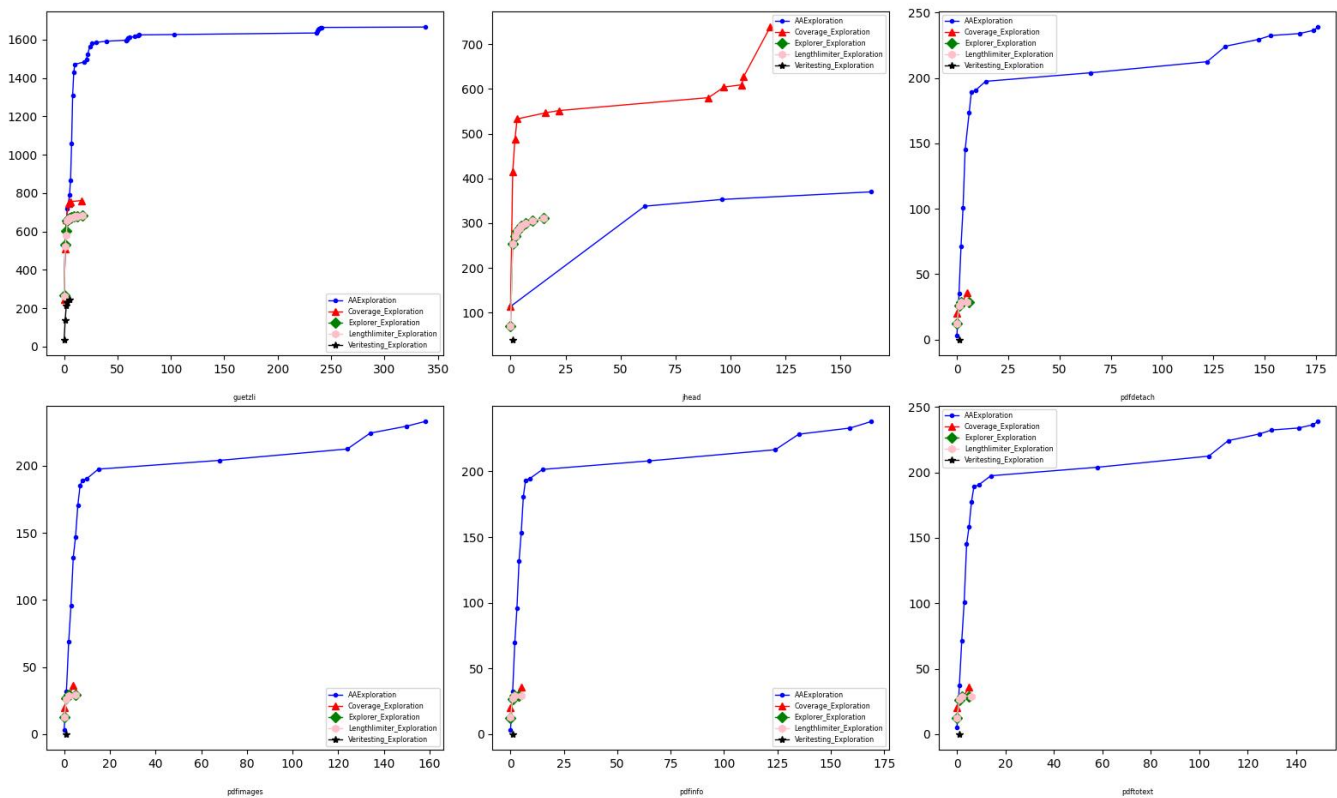
### 5.3. Path Exploration Capability Improvement

After analyzing the performance overhead and the effectiveness of state space explosion mitigation by EtWExplorer's abstract syntax tree analysis technique. The core goal of EtWExplorer is to explore more paths during the symbolic execution. If EtWExplorer's multi-priority scheduling technique based on abstract syntax tree analysis causes a loss of path exploration capability. Then the previous performance tests and state space explosion mitigation will become meaningless. Therefore, whether EtWExplorer does not lose the accuracy of path exploration with multi-priority scheduling and pruning of states in symbolic execution is the third question we want to test. Here, we use the same test program as in Section 5.2.

In Figures 5 and 6, the x-axis represents the explore technology execution time in minutes, and the y-axis represents the number of basic blocks found (Figure 5) and edges found (Figure 6). The meaning represented by the lines of each color in the figure is the same as in Figure 3. We will make one note here. Since Veritestng requires the construction of a local control flow graph, the precise local control flow graph is difficult to be constructed. Therefore, the path exploration performance of Veritestng performs weakly in the experiments.



**Figure 5.** Improvements in path exploration capability (block coverage) when exploring programs that cause state space explosions.



**Figure 6.** Improvements in path exploration capability (edge coverage) when exploring programs that cause state space explosions.

As we can see from the graphs, many exploration techniques exit just at the beginning of the test due to the state space explosion which we tested in Section 5.2. When there is a state space explosion, the symbolic execution engine needs to perform symbolic execution for each state, which severely affects the efficiency of symbolic execution and thus makes it difficult to discover new paths. At the same time, the large amount of state space leads to an increase in memory usage. Then the memory usage reaches the 20 GB we set, the symbolic execution will exit directly. Therefore, in this experiment, not all path exploration techniques end at the same time. Path exploration is also constrained by memory usage.

Finally, we calculated a mean value by adding up the results obtained from the six procedures. The path exploration capability (number of unique basic blocks found) ratio is the EtWExplorer technique: Coverage-based technique: Angr’s Explorer technique: Lengthlimiter technique: Veritestig technique = 2489: 1246: 921: 921: 253. The path exploration capability (Number of unique edges found) ratio is EtWExplorer technique: Coverage-based technique: Angr’s Explorer technique: Lengthlimiter technique: Veritestig technique = 2984: 1643: 1111: 1111: 285. EtWExplorer has a 199% to 983% improvement in the path exploration capability of block coverage and a 181% to 1047% improvement in path exploration capability of edge coverage. We can find a significant improvement in EtWExplorer’s path exploration capability in programs that would cause a state space explosion.

In summary, EtWExplorer can significantly improve the path exploration capability of symbolic execution techniques in programs that cause path explosion while introducing an acceptable range of performance overhead. This has an important meaning in vulnerability discovery.

## 6. Discussion

Our core work proposes a novel technique for path exploring in symbolic executions. EtWExplorer performs multi-priority scheduling based on abstract syntax tree analysis and mitigates the impact of state space explosion on path exploration in symbolic executions.

We analyzed that the cause of the state space explosion during symbolic execution is the massive growth of the state space caused by branching during repeated code execution. How to determine whether repeated code should be symbolically executed is the key problem to solving the state space explosion.

In this paper, the path exploration capability of symbolic execution is taken as the primary goal of our study. Therefore, we use the approach of abstract syntax tree analysis to relate the expressions generated during the symbolic execution of basic blocks to unsatisfiable constraints. The execution state is prioritized and pruned by the relationship between the variation of the expressions generated during repeated execution and the unsatisfiable constraints. This is used to mitigate the state space explosion encountered during symbolic execution.

We experimented with EtWExplorer and obtained the following conclusions.

1. EtWExplorer introduces a performance overhead of 72% in the worst case and can improve performance by 294% in the best case;
2. EtWExplorer has a 95% improvement in the state space explosion mitigation capability when facing programs that cause a state space explosion;
3. EtWExplorer has a 199% to 983% improvement in the path exploration capability of block coverage and a 181% to 1048% improvement in the path exploration capability of edge coverage when facing programs that cause a state space explosion.

## 7. Related Work

The current stage of symbolic execution research focuses on the following domains.

To enhance the applicability of symbolic execution, intermediate representations are widely used. By lifting the binary code into an intermediate representation, symbolic execution tools can analyze programs of different architectures (e.g., x86, x86-64, ARM, MIPS). This enhances the reusability of symbolic execution tools so that it is not restricted to a specific platform. KLEE [11] uses the LLVM compiler to lift C or C++ code to bytecode and parse on it. Valgrind's [12,13] dynamic analysis framework presents an intermediate representation of VEX, which uses a RISC-like set of ideas designed for program analysis. It applies to both 32-bit and 64-bit ARM, MIPS, PPC, and x86 binaries. Angr is also implemented through VEX IR. In *S<sup>2</sup>E* [14], the authors implemented an x86 to LLVM lifter that assists KLEE in symbolic execution in a whole-system virtualized environment. This research further enhances the applicability of symbolic execution.

Constructing a more accurate control flow graph (CFG) to aid in the program analysis is also an important function of symbolic execution. Value-set analysis (VAS) can effectively identify program state properties (e.g., indirect jumps) with a tight over-approximation. BitBlaze [15] inserts specific successor nodes into the indirect jumps when initializing the CFG. The value-set analysis is then used when a more accurate control flow graph is required. Angr provides two different algorithms for generating CFG: the "fast" method uses static analysis to analyze the basic blocks in the program; the "emulation" method obtains the return address of the basic block by emulating the execution of the basic block.

Being against malicious obfuscation techniques is also a research point in the field of symbolic execution. Some malicious code obfuscations prevent security researchers from understanding their internal logic.

BE-PUM [16] uses concolic execution [17] for adversarial code obfuscation.

## 8. Conclusions

In this paper, we propose a multi-priority scheduling path exploration technology based on abstract syntax tree analysis. This approach can effectively mitigate the state space explosion during symbolic execution. It effectively improves the path exploration capability of symbolic execution with less performance overhead. We implemented our proposed approach in EtWExplorer and achieved significant results in real-world programs.

**Author Contributions:** Conceptualization, X.H.; methodology, X.H.; software, X.H.; validation, X.H.; investigation, X.H.; writing—original draft preparation, X.H.; writing—review and editing, X.H. and P.W.; supervision, P.W.; funding acquisition, P.W., K.L. and X.Z. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported by the Natural Science Foundation of China (61902412, 61902405, 62272472)), the National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013), the Natural Science Foundation of Hunan Province of China under grant no. 2021JJ40692, and the Research Project of the National University of Defense Technology (ZK20-17, ZK20-09).

**Acknowledgments:** We sincerely thank the reviewers for their insightful comments that helped us improve this work.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Abbreviations

The following abbreviations are used in this manuscript:

AST	abstract syntax tree
CTF	capture the flag
CFG	control flow graph

## References

- Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting Fuzzing through Selective Symbolic Execution. In Proceedings of the NDSS, San Diego, CA, USA, 21–24 February 2016.
- Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
- Chen, Y.; Li, P.; Xu, J.; Guo, S.; Zhou, R.; Zhang, Y.; Wei, T.; Lu, L. Savior: Towards bug-driven hybrid testing. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1580–1596.
- Kim, K.; Jeong, D.R.; Kim, C.H.; Jang, Y.; Shin, I.; Lee, B. HFL: Hybrid Fuzzing on the Linux Kernel. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2020.
- Huang, H.; Yao, P.; Wu, R.; Shi, Q.; Zhang, C. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1613–1627.
- Ma, K.K.; Yit Phang, K.; Foster, J.S.; Hicks, M. Directed symbolic execution. In Proceedings of the International Static Analysis Symposium, Venice, Italy, 14–16 September 2011; pp. 95–111.
- Majumdar, R.; Sen, K. Hybrid concolic testing. In Proceedings of the 29th International Conference on Software Engineering (ICSE’07), Minneapolis, MN, USA, 20–26 May 2007; pp. 416–426.
- Shoshitaishvili, Y.; Wang, R.; Salls, C.; Stephens, N.; Polino, M.; Dutcher, A.; Grosen, J.; Feng, S.; Hauser, C.; Kruegel, C.; et al. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016.
- Avgerinos, T.; Rebert, A.; Cha, S.K.; Brumley, D. Enhancing symbolic execution with veritesting. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 1083–1094.
- Moura, L.D.; Bjørner, N. Z3: An efficient SMT solver. In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, 29 March–6 April 2008; pp. 337–340.
- Cadar, C.; Dunbar, D.; Engler, D.R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the OSDI, San Diego, CA, USA, 8–10 December 2008; Volume 8, pp. 209–224.
- Nethercote, N.; Seward, J. Valgrind: A program supervision framework. *Electron. Notes Theor. Comput. Sci.* **2003**, *89*, 44–66. [[CrossRef](#)]
- Nethercote, N.; Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Not.* **2007**, *42*, 89–100. [[CrossRef](#)]
- Chipounov, V.; Kuznetsov, V.; Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM Sigplan Not.* **2011**, *46*, 265–278. [[CrossRef](#)]
- Song, D.; Brumley, D.; Yin, H.; Caballero, J.; Jager, I.; Kang, M.G.; Liang, Z.; Newsome, J.; Poosankam, P.; Saxena, P. BitBlaze: A new approach to computer security via binary analysis. In Proceedings of the International Conference on Information Systems Security, Patna, India, 16–20 December 2008; pp. 1–25.
- Hai, N.M.; Ogawa, M.; Tho, Q.T. Obfuscation code localization based on CFG generation of malware. In Proceedings of the International Symposium on Foundations and Practice of Security, Paris, France, 7–10 December 2015; pp. 229–247.
- Sen, K. Concolic testing. In Proceedings of the Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, Georgia, AL, USA, 5–9 November 2007; pp. 571–572.