VITAL: Vulnerability-Oriented Symbolic Execution via Type-Unsafe Pointer-Guided Monte Carlo Tree Search

HAOXIN TU, Singapore Management University, Singapore
LINGXIAO JIANG, Singapore Management University, Singapore
MARCEL BÖHME, Max Planck Institute for Security and Privacy, Germany

How do we find new memory safety bugs effectively when navigating a symbolic execution tree that suffers from the well-known path explosion challenge? Existing solutions either adopt path search heuristics to maximize coverage rate or chopped symbolic execution to skip uninteresting code (i.e., manually labeled as vulnerability-unrelated) during path exploration. However, most existing search heuristics are not vulnerability-oriented, and manual labeling of irrelevant code-to-be-skipped relies heavily on prior expert knowledge, making it hard to detect vulnerabilities effectively in practice.

This paper proposes VITAL, a new vulnerability-oriented path exploration for symbolic execution with two innovations. First, a new indicator (i.e., *type-unsafe* pointers) is suggested to approximate vulnerable paths. A pointer that is *type-unsafe* cannot be statically proven to be safely dereferenced without memory corruption. Our key hypothesis is that a path with more *type-unsafe* pointers is more likely to be vulnerable. Second, a new *type-unsafe* pointer-guided Monte Carlo Tree Search algorithm is implemented to guide the path exploration towards the areas that contain more *unsafe* pointers, aiming to increase the likelihood of detecting vulnerabilities.

We built VITAL on top of KLEE and compared it with existing path searching strategies and chopped symbolic execution. In the former, the results demonstrate that VITAL could cover up to 90.03% more unsafe pointers and detect up to 57.14% more unique memory errors. In the latter, the results show that VITAL could achieve a speedup of up to 30x execution time and a reduction of up to 20x memory consumption to detect known vulnerabilities without prior expert knowledge automatically. In practice, VITAL also detected one previously unknown vulnerability (a new CVE ID is assigned), which developers have fixed.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Reliability, Security, Software testing, Program analysis, Symbolic execution.

ACM Reference Format:
Haoxin Tu, Lingxiao Jiang, and Marcel Böhme. 2025. VITAL: Vulnerability-Oriented Symbolic Execution via Type-Unsafe Pointer-Guided Monte Carlo Tree Search. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 111 (June 2025), 24 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Memory unsafety is the leading cause of vulnerabilities in complex software systems [8, 15, 49, 53]. For example, based on a report from Microsoft, more than 70% of security bugs have been memory safety problems in the last 12 years [15]. To prevent memory errors, many advanced static/dynamic/symbolic analysis-based approaches are devoted to detecting such errors automatically. Among them, *symbolic execution* is considered one of the promising program analysis techniques [9, 52, 65]. The key idea of symbolic execution is to simulate program executions with symbolic

Authors' Contact Information: Haoxin Tu, Singapore Management University, Singapore, haoxintu0@gmail.com; Lingxiao Jiang, Singapore Management University, Singapore, lxjiang@smu.edu.sg; Marcel Böhme, Max Planck Institute for Security and Privacy, Germany, marcel.boehme@acm.org.

Manuscript submitted to ACM

inputs and generate test cases by solving path constraints collected during execution. Taking advantage of the soundness of test case generation, symbolic execution has also been applied in many other areas, such as programming language [31, 34, 46] and security [1, 11, 74].

One of the key challenges in symbolic execution is path explosion, where even a small program can produce a vast execution tree [2]. Two main alternatives have been proposed to alleviate this challenge. The first alternative is to steer the exploration of the path to maximize the rate at which the new code is covered using various search heuristics. Notably, the symbolic execution KLEE [9] supports random, breadth-first (`bfs`), depth-first search (`dfs`), and other coverage-guided heuristics. A recent work CBC [71] proposes to use compatible branch coverage-driven path exploration for symbolic execution. Cgs [57], FeatMaker [73], and Empc [69] are the most recent works that propose new path search heuristics to improve code coverage. However, existing studies show that achieving the best coverage does not necessarily mean that the largest number of vulnerabilities can be detected [5, 13].

Another solution is to skip the symbolic execution of certain code that is manually labeled as not related to the vulnerability using *chopped symbolic execution* [61]. However, setting up chopped execution requires prior expert knowledge of the program under test and intensive manual effort to decide which functions to skip. For example, to successfully detect a vulnerability (CVE-2015-2806) in the `libtasn1` library, users need to locate four specific functions and two lines to skip[1]. Since the library includes more than 20,000 lines of code, locating these functions/lines may require expert knowledge and involve intensive human efforts. Also, it is extremely difficult to use it to find new vulnerabilities, as we do not know where the vulnerabilities are until they are caught. Another issue with chopped executions is their performance. It consumes significant memory when switching between skipped and normal execution (see more details in Section 4.2). Motivated by addressing the above limitations, we aim to investigate the following research question: *How can we automatically conduct vulnerability-oriented path exploration to detect new vulnerabilities without relying on prior expert knowledge?*

Conducting vulnerability-oriented path exploration is non-trivial and presents challenges for at least the following two reasons. First, we should decide which indicators can effectively approximate the vulnerability of a path. To our knowledge, no existing unified metrics could be used to indicate a path containing memory errors (e.g., *out-of-bounds*). Therefore, it is challenging to find a suitable indicator to represent vulnerable paths. Second, it is also difficult to effectively leverage indicators to guide path exploration. To make path exploration more effective, we argue that a promising path search strategy should maintain a good trade-off between exploiting the paths that have already been executed in the past and exploring the paths that will be executed in the future. However, existing search heuristics do not fully take advantage of past execution, which degrades the possibility of exploring likely vulnerable paths.

In this paper, we propose Vital[2], a new **V**ulnerability-or**I**en**T**ed p**A**th exp**L**oration strategy for symbolic execution via type-unsafe pointer-guided Monte Carlo Tree Search (MCTS). Our core insight is that spatial memory safety errors (e.g., *out-of-bounds* memory accessing errors) can *only* happen when dereferencing *type-unsafe* pointers, i.e., pointers that cannot be statically proven to be memory safe [30, 43, 59]. As shown in Figure 1, where the data is from in our experiments (in Section 4.1), we found that an increase in the number of unsafe pointers exercised is directly related to an increasing number of memory errors, with a strong positive Pearson coefficient 0.93 [16]. Hence, we suggest maximizing the number of unsafe pointers during path exploration to address the first challenge of approximating vulnerability. To address the second challenge of effective search within the execution tree, we drive a new pointer-guided MCTS that effectively balances the path exploration and exploitation to maximize the number of covered unsafe pointers.

[1]Manually defined option used in `Chopper`: "*–skip=__bb0,__bb1,_asn1_str_cat:403/404,asn1_delete_structure*"
[2]The name also reflects our aim for exploring *vital* program paths due to path explosion.

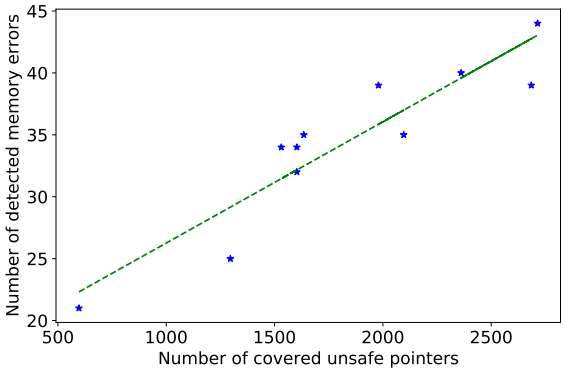


Fig. 1. Correlation between unsafe pointers and memory errors (Pearson's coefficient [16]: 0.924)

We implement VITAL on top of KLEE [9] and conduct extensive experiments to compare VITAL with existing search strategies and chopped symbolic execution. We run VITAL against six path exploration strategies in KLEE and a recently proposed strategy, CBC [71] on the GNU `Coreutils`, and the results demonstrate that VITAL could cover up to 90.03% more unsafe pointers and detect up to 57.14% more unique memory errors. Further, we run VITAL against KLEE [9], `Chopper` [61], and CBC [71] over six known CVE vulnerabilities, and the results show that VITAL could achieve a speedup of up to 30x execution time and a reduction of up to 20x memory consumption to detect all of them automatically without prior expert knowledge. VITAL also detected one previously unknown vulnerability missed by other approaches. It has been confirmed and fixed by developers, with a new CVE ID assigned.

In summary, this paper makes the following contributions.

- To our knowledge, VITAL is the first work performing vulnerability-oriented path exploration for symbolic execution towards effective and efficient vulnerability detection.
- We suggest a new indicator (i.e., the number of type-unsafe pointers) to approximate the vulnerability (or vulnerability-proneness) of a program path and utilize a new type-unsafe pointer-guided Monte Carlo Tree Search algorithm to navigate smart vulnerability-oriented path searches.
- Extensive experiments demonstrate the superior performance of VITAL in terms of unsafe pointer coverage and memory errors/vulnerability detection compared to state-of-the-art approaches. In practice, VITAL also detected a previously unknown vulnerability, indicating its practical vulnerability detection capability.
- We publish the replication package online [63], to foster practical symbolic execution for vulnerability detection.

Organizations. Section 2 gives the background and motivation. Section 3 describes the design of VITAL. Section 4 presents our evaluation results. Section 5 discusses the overhead of type inference, the impact of different configurations, and threats to validity. Sections 6 and 7 describe related works and conclude with future work. We also provide extra discussions and experimental results in the Appendix in the Supplementary Material or online at Appendix.

2 Background and Motivation

2.1 Background

2.1.1 Symbolic Execution and Path Exploration. Symbolic execution is a program analysis technique that analyzes a test program by feeding the program with symbolic inputs. During symbolic execution, path constraints are collected,

and corresponding test cases will be generated by off-the-shelf constraint solvers (e.g., STP [54] or Z3 [75]). The main activity in symbolic execution is to consistently select a path (or state) to explore and analyze an instruction one time until no state remains or a given timeout is reached. Notably, the widely used KLEE symbolic execution engine, the representative search strategies include `bfs`, `dfs`, `random`, code coverage-guided (i.e., `nurs:covnew`), and instruction coverage-guided (i.e., `nurs:md2u` and `nurs:icnt`). Recent work CBC [71], Cgs [57], FeatMaker [73], and Empc [69] also propose new search strategies to improve path exploration.

2.1.2 Memory Safety and Type Inference. Existing memory safety vulnerabilities fall mainly into two main categories: *spatial* and *temporal* memory safety vulnerabilities [30, 65]. The first ones occur when pointers reference addresses outside the legitimate bounds (e.g., *buffer overflow*), and *temporal* memory safety issues arise from the use of pointers outside of its live period (e.g., *use-after-free*). Previous studies [42, 76] show that serious vulnerabilities caused by violating spatial safety are well known in the community. For example, more than 50% of recent CERT warnings are due to breaches of spatial safety [66]. Furthermore, compared to temporal safety issues, spatial safety vulnerabilities can be exploited by many mature techniques, such as return-oriented programming [48] and code reuse attacks [4, 12]. The above fact indicates that designing advanced solutions to keep spatial memory safe is of critical importance.

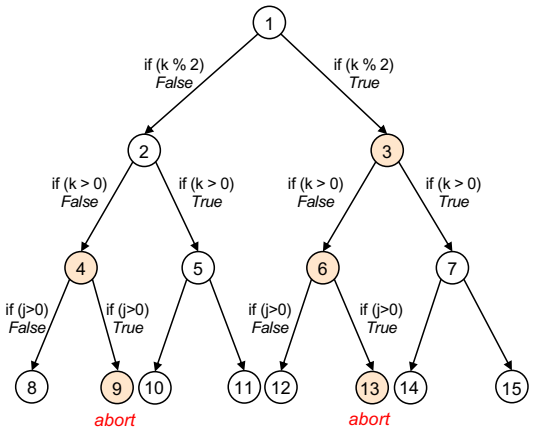
To enforce complete spatial safety, the only way is to keep track of the pointer bounds (the lowest and highest valid address to which it can point) [59]. Integrating a type inference system by static analysis into a type-unsafe programming language (e.g., C) is a well-established strategy for keeping track of pointer bounds [22, 30] to prove memory safety. For example, `CCured` [43] statically infers pointer types into three categories: *SAFE*, Sequence (*SEQ*), and Dynamic (*DYN*). The *SAFE* pointers are the dominant portion of the program (e.g., 92.29% of the pointers are safe reported by a recent work [30]) that can be proved to be free of memory errors [43]. The *SEQ* pointers require extra bound checking, and *DYN* pointers need to perform run-time checks to ensure memory safety. Since spatial memory error can only occur within *SEQ* or *DYN* pointer categories, we refer to *SEQ* or *DYN* pointers as *type-unsafe* or *unsafe* pointers in this paper. Taking the code from Figure 2(a) for example, the pointers `p.y` (Line 9), `p.z` (Line 16), and `p.y` (Line 20) are used in pointer arithmetic, which are classified as *unsafe* (*SEQ*) pointers. Such classification results can be obtained with mature type inference tools such as Ccured [30].

2.1.3 Monte Carlo Tree Search (MCTS). MCTS is a heuristic search algorithm to solve decision-making problems, especially those in games (e.g., AlphaGo) [6, 40]. The fundamental idea behind MCTS is to use reward-weighted randomness to simulate decision sequences and then to use the results of these simulations to decide the most promising move. It excels at balancing between exploring new moves in the future and exploiting known good moves in the past. Technically, the MCTS consists of the following four steps ([6]):

- *Selection*: starting from the root node, select successive child nodes that are already in the search tree according to the *tree policy* down to an expandable node that has unvisited children.
- *Expansion*: expand the unvisited node into the search tree.
- *Simulation*: perform a playout from the expanded node governed by a *simulation policy*. The termination of the simulation yields results (i.e., rewards) based on a reward function.
- *Backpropagation*: update the information stored in the nodes on the path from the expanded node to the root node with the calculated reward. This also includes updating the visit counts.

```
1  struct point {int x, y, z;};
2
3  int main() {
4    struct point p = {0, 0, 0};
5    int j, k;  // j and k are symbols
6
7    f(&p, k);
8    if (j > 0){
9      if (p.y) abort();// unsafe pointer
10   } else
11       ;
12   return 0;
13 }
14 void f(struct point * p, int k){
15     if (k % 2)
16         p->z++; // unsafe pointer
17     if (k > 0)
18         p->x++;
19     else
20         p->y++; // unsafe pointer
21 }
```

(a) Code Example



(b) Execution tree of code example

Fig. 2. Motivating example (the code example is adapted from `Chopper` [61])

Note that the *tree policy* (including node selection and expansion) and *simulation policy* are two key factors in conducting effective MCTS applications. Since then, a large number of enhancements have been proposed to facilitate the capabilities of MCTS [6, 58].

2.2 Motivating Example

To illustrate the motivation behind VITAL, we will use a simple example shown in Figure 2(a) (adapted from `Chopper` [61]) to show the limitations of the existing solution and the advantage of VITAL. Figure 2(b) represents the corresponding process tree, where the arrow denotes the execution flow with branch conditions and each circle represents an execution state. The colored nodes mean they include unsafe pointers, where the pointers are code blocks that are unique to two forked states. For example, when the executor forks at Line 17, the code in Lines 18 and 19 is two unique code blocks for the two forked states. The eight leaf nodes are terminal states, while the others are internal states. Assuming that we are trying to answer the following question: *How can we efficiently trigger the problematic* `abort` *failure at Line 9?*

Direction 1: Path Search Heuristics. Two commonly used search strategies are `bfs` and `dfs`, both of which apply a fixed order to explore all program paths in the following ranked order, where the leaf nodes represent the termination states (the "->" represents the order of terminated states):

- `bfs`: (15) -> (14) -> (13) -> (12) -> (11) -> (10) -> (9) -> (8)
- `dfs`: (8) -> (9) -> (10) -> (11) -> (12) -> (13) -> (14) -> (15)

We can observe that both of them can not explore (or rank) the path containing the `abort` failure at the top-1 position, so they might not be efficient. A random search might first find the `abort` path, but the result is unreliable. Other search heuristics, such as coverage-guided (e.g., CBC [71]), are good at covering new code but are still likely to miss certain errors because the best coverage has no statistical relation with the largest number of bugs [5].

Direction 2: Chopped Symbolic Execution. An alternative solution to detect failure efficiently is to skip *uninteresting* functions (function `f` in the example) before execution. When the execution goes to the statement that depends on the skipped function (i.e., Line 9), it recovers the execution of function `f` and merges the remaining states to avoid unsound results. However, chopped execution relies heavily on the pre-defined skipped functions, which require prior

expert knowledge to obtain. Furthermore, the recovery mechanism, which switches between recovering and normal execution, is memory-consuming (see more details in Section 4.2).

Our Solution: VITAL. Unlike existing solutions, VITAL performs vulnerability-oriented path search. The basic idea behind VITAL is that, before execution, we obtain the type-unsafe pointer locations (e.g., Lines 9, 16, and 20), where the type-unsafe pointers approximate the existence of vulnerabilities (i.e., the `abort` failure in this example). Then, guided by the locations, VITAL will give a higher reward to those states with unsafe pointers, navigating the exploration towards the path where the number of unsafe pointers is maximized, i.e., (1) -> (3) -> (6) -> (13). As a result, VITAL explores the state (13) that triggers the `abort` failure at the top-1 position.

It is worth noting that although the existing search strategies in *Direction 1 & 2* can ultimately trigger the `abort` failure in the tiny example, due to the large spaces to explore in more realistic path exploration over complex software systems, they are very likely to miss important vulnerabilities due to inefficiency. In contrast, VITAL could effectively explore the most promising paths that are more likely to contain vulnerabilities in practice.

3 Approach

Overview. The general procedure of VITAL is to continuously select a promising state that is more likely to contain vulnerabilities. Technically, VITAL first acquires a set of unsafe pointers by performing pointer type inference over the test program only once at compile-time. After that, guided by unsafe pointers, VITAL corporates a new search strategy (i.e., MCTS) to have an optimal balance between the exploration of future states and the exploitation of past executed states at execution time. The main technical contribution of VITAL lies in a new symbolic execution engine that implements a new path exploration strategy, based on a variant of MCTS equipped with the unsafe pointer-guided node expansion and the customized simulation policy, aiming to detect new vulnerabilities in software systems.

3.1 Acquisition of Type-Unsafe Pointers via Type Inference System

This subsection first justifies why the type-unsafe pointer is a reasonable indicator to approximate unknown memory safety vulnerability, and then articulates how VITAL collects them.

3.1.1 Why Type-Unsafe Pointers? The main root cause of memory safety problems in C/C++ is due to sacrificing type safety for flexibility and performance in the early design choice in the 1970s [43]. As mentioned in Section 2.1.2, to ensure spatial memory safety, it is essential to track specific properties (e.g., size and types) of the memory area to which the pointer refers. A type inference system is a well-established static analysis technique that was used to keep spatial memory safe in the literature [22, 59]. For example, `CCured` [43] keeps spatial memory safe by classifying pointer types based on the usage of the pointers, while *SAFE* pointers that are free of memory errors can be soundly determined at compile time. For others (i.e., *SEQ* or *DYN*), memory safety must be ensured at run-time, which requires the insertion of safety checks during execution.

Since there are no unified indicators to approximate unknown memory safety vulnerabilities, we suggest leveraging pointer types (i.e., *SEQ* and *DYN*) that can not be statically verified to be free of memory errors as *unsafe* pointers and use them to approximate vulnerable behaviors. We assume that if a program path contains more unsafe pointers, the path is more likely to contain vulnerabilities (the results shown in Figure 1 also support our claim).

3.1.2 How Does VITAL Acquire Type-Unsafe Pointers? We follow an existing type inference algorithm `CCured` [30] to classify pointer types according to the following three rules:

(1) All pointers are classified as *SAFE* upon their declaration.

Algorithm 1: Unsafe Pointer-guided MCTS State Selection

```
   Input: unsafe pointer set unsafeSet, a current state cur_state
   Output: an execution state to be executed next n_state
1  Function MCTSSearch::selectState():
2      ExecutionState n_state
3      while (! isTerminal(cur_state->node)) do
4          if (! hasEligibleChildren(cur_state->node)) then
5              n_state = doSelection(cur_state->node)
6              cur_state = n_state
7          else
8              n_state = doExpansion(cur_state->node)
9              if (isWorthSimu(cur_state->node)) then
10                 reward=doSimulation(n_state->node,unsafeSet)
11                 doBackpropagation(reward, n_state->node)
12             break
13     return n_state
```

(2) *SAFE* pointers that are subsequently used in pointer arithmetic are re-classified as *SEQ*.

(3) *SAFE* or *SEQ* pointers that are interpreted with different types are re-classified as *DYN*.

The above design helps yield conservatively overestimated unsafe pointers, meaning that there are no pointers classified as *SAFE* but used in an *unsafe* way [41, 43]. In other words, VITAL may potentially identify *non-DYN* pointers as *DYN* pointers, but never misclassify *DYN* pointers as *non-DYN*.

3.2 Type-unsafe Pointer-guided Monte Carlo Tree Search

The goal of a search strategy in symbolic execution is to select an interesting state to execute next. Algorithm 1 shows the overall selection strategy designed in VITAL. It takes a set of unsafe pointers *unsafeSet* acquired from the previous step and the current execution state being executed *cur_state*, and outputs the expected state to be executed next. There are many tree search algorithms proposed in the literature, such as Greedy Best-First Search [29], Bidirectional Search [55], Uniform Cost Search [14], and MCTS [6]. We chose MCTS in this study because it excels at maintaining a good trade-off between exploring new states in the future and exploiting known states in the past. The overall workflow of Algorithm 1 customizes the standard MCTS algorithm with a few key steps guided by the unsafe pointer set. Before diving into the details of the algorithm, we want to clarify that we use the process tree (internal data structure supported in KLEE [9]) as the symbolic execution tree to be used for MCTS. Since every node represents an execution state in the process tree, we will use the term node and state interchangeably in the following sections.

The algorithm starts by performing a *while-loop* to check if the current state *cur_state* is a terminal state or not (Line 3). If not, it checks whether the current state has eligible (i.e., expandable tree node) children via function `hasEligibleChildren` (Line 4). The result will be either going back to the *while-loop* after performing node selection via `doSelection` (Algorithm 2) in the *if-true* branch (Line 5) or calling `doExpansion` (Algorithm 3) in the *if-false* branch (starting at Line 8). After the node expansion, it simulates the expanded node. It gets a reward by invoking `doSimulation` (Algorithm 4 at Line 10) if the function `isWorthSimu` return *true* (Line 9). A node is worth simulating when it has never been simulated or does not reach the simulation limit to avoid an infinite loop. Later, the reward is backpropagated through `doBackpropagation` function to all the parent nodes until the root. Finally, the expanded

state is returned as normal (Line 13). It is worth noting that some key steps, such as node expansion and simulation, take actions based on the unsafe pointer set as one of the function parameters, where the record of unsafe pointers is essential to guide the smart search process in MCTS. We will explain each step in the following.

3.2.1 Tree Node Selection. The goal of the tree node selection is to select a child node that is already in the search tree. Algorithm 2 shows the overall procedure of node selection. Given the input of a tree node, it checks whether both left and right nodes exist in the search tree. If both nodes are valid, it selects the best child by calling `selectBestChild` (Line 3). Otherwise, only the valid left or right node will be selected (Lines 4-7).

Inside function `selectBestChild`, the child node is selected based on the highest UCT (i.e., Upper Confidence bounds applied to Trees) value. UCT is a widely recognized algorithm that addresses a significant limitation of MCTS [6, 33, 38], where the MCTS may incorrectly favor a suboptimal move that has a limited number of forced refutations. The UCT formula used to balance the exploitation and exploration is defined as $UCT(s, s') = \frac{R(s')}{V(s')} + C\sqrt{\frac{2 \ln V(s)}{V(s')}}$ (details can be found in Section 3.3 in [6]), where s is the current state, s' is the child state of state s being selected, $V(s)$ indicates how many times the state has been visited, and $R(s)$ is the cumulative reward of all the simulations that have passed through this state. C is a constant that controls the degree of exploration.

Algorithm 2: Procedure of Tree Node Selection in VITAL

```
   Input: a tree node node (with attributes such as isInTree)
   Output: a leaf node of node to be selected selected_node
1  Function doSelection(node):
2      if node->left->isInTree && node->right->isInTree then
3          selected_node = selectBestChild(node)
4      if node->left->isInTree && ! node->right->isInTree then
5          selected_node = node->left
6      if ! node->left->isInTree && node->right->isInTree then
7          selected_node = node->right
8      return selected_node
9  Function selectBestChild(node):
       /* return the node with the highest UCT value                    */
```

When a node is selected (Line 8), we check whether the node has any successors that have not yet been added to the search tree. If such a child exists, `doExpansion` is performed; otherwise, if the node is fully expanded, the algorithm returns to the while-loop to continue selection from the best child node of the successors.

3.2.2 Tree Node Expansion. The expansion aims to expand a child that is *not* in the search tree yet.

Algorithm 3 presents the overall procedure of tree node expansion. Given an input node, different from node selection, it checks whether both left and right nodes do not exist in the search tree. If neither node is in the search tree, it expands the best child by calling `expandBestChild` (Line 3). Otherwise, only the left or right node will be augmented for expansion (Lines 4-7). Inside the function `expandBestChild`, the node is expanded based on the highest expansion score (*ExpScore*) among the two branches.

To obtain the score of two branches after an execution state is forked, three steps are performed: (1) collecting unique program elements, i.e., basic blocks, that are *unique* to each state; (2) analyzing the collected basic blocks and finding a way to weigh which state is more vulnerable; and (3) scoring the two branches based on the weighting measurement.

Algorithm 3: Procedure of Tree Node Expansion in VITAL

```
Input: a tree node node, (global) unsafe pointer set unsafeSet
Output: a leaf node of node to be expanded expanded_node
1 Function doExpansion(node):
2     if ! node->left->isInTree && ! node->right->isInTree then
3         expand_node = expandBestChild(node, unsafeSet)
4     if node->left->isInTree && ! node->right->isInTree then
5         expanded_node = node->right
6     if ! node->left->isInTree && node->right->isInTree then
7         expanded_node = node->left
8     return expanded_node
9 Function expandBestChild(node, unsafeSet):
      /* return the node with the highest ExpScore value                    */
```

To accomplish the first step, we use the dominance relationships of graph theory [39] on the Control Flow Graph (CFG) of a function to collect the unique basic blocks for each state. The idea is to find the *post-dominator* of the two new forked states and to collect the basic blocks in between each of the states and their post-dominator; such basic blocks indicate how the two states may induce different executions. A node A on a graph is said to *dominate* another node B if every path from the graph's entry point (start node) to B must go through A. Conversely, a node A is a *post-dominator* of another node B if every path from B to the exit point (or end node) of the graph must pass through A. Taking the function `f` in the example shown in Figure 2 as an illustration, the CFG of the function is shown in Figure 3. Starting from the entry block (`BB1`), the execution will go to Line 15 and the engine will fork two states (`TrueState` and `FalseState`), where the *pc* (points to the next execution instruction) `TrueState` points to the basic block (`BB2`) starting at Line 16. The *pc* in `FalseState` points to the basic block (`BB3`) after the *if-branch* at Line 15. The unique basic block for `TrueState` is `BB2`, as both states will go through the post-dominator block `BB3`. For the following forking at Line 17 in Figure 2(a), again, the unique basic block for two states is `BB4` and `BB5`, respectively.

Then, following the intuition that more unsafe pointers in a path implies more vulnerabilities, we use the number of unsafe pointers in the basic blocks to quantify the expansion scores of the states: $ExpScore(s) = N_{unsafe}(s)$, where s is a state being scored and $N_{unsafe}(s)$ represents the number of unsafe pointers in the basic blocks collected as above for s.

3.2.3 Tree Node Simulation. Simulation is a crucial component of the MCTS algorithm as it enables the evaluation of non-terminal nodes by performing random playouts (i.e., simulation runs) to estimate the potential outcomes (i.e., rewards) of different actions. This process helps balance exploration and exploitation by providing statistical sampling to approximate the reward of execution states, which guides the algorithm in making promising decisions.

There can be many options to perform each simulation run. Intuitively, for our symbolic execution, we could consider the following three options (OPs) to get the reward of a node.

- *OP1*: statically analyze a single path following the state based on control-flow graphs.
- *OP2*: symbolically run a single feasible path following the state with a fixed number of executed instructions.
- *OP3*: symbolically run a single feasible path following the state until the simulated state terminates.

Note that a high-quality simulation policy should be efficient, have few false positives, and have a certain degree of randomness [6, 38]. As shown in the literature [6], a certain degree of randomness is advantageous as it is simple and requires no domain knowledge, which will most likely cover diverse areas of a search space.

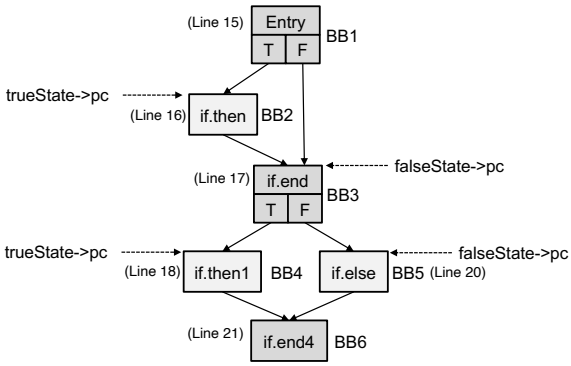


Fig. 3. CFG of function f shown in Figure 2(a) illustrating how to get the score of true/false states

Concerning the above criteria, *OP1* might be good in terms of randomness, but efficiency and false positives may be an issue: Computing CFGs may be costly, and paths in static CFGs may be infeasible. For *OP2*, it may comply with the criteria, but it might be difficult to decide what is the preferable number of instructions to use when evaluating the reward of a node across different runs. *OP3* could be better as it terminates based on the behavior of program execution. Technically, the simulation execution is essentially a lightweight depth-first execution of a single random feasible path. When a node is expanded, a new node is forked out for simulation. The new node's path constraints remain, and the simulation is started by setting the node attribute "*inhibitForking=true*" to prevent further forking. Then, whenever a branch condition is encountered during simulation, a feasible random branch is chosen until the execution terminates. We empirically evaluate different fixed instruction numbers for *OP2*, and the results show that they are inferior compared with *OP3* (see more details in Section 4.3).

Algorithm 4 shows the simplified procedure of the simulation process designed in Vital. It takes a tree node to be expanded and the unsafe pointer set as input, and yields a reward after simulation. Inside the algorithm, a vector of basic blocks *simulatedBB* is first initiated (Line 2). The vector stores all basic blocks executed during simulation via the function `executor.sim_run`. Then, two metrics (the number of unsafe pointers *nu_unsafe* and memory errors *nu_error*) are calculated (Line 4) or updated (Line 5), which will be used to get the reward of this simulation run (Line 6). In this study, Vital is novel in utilizing the random and lightweight simulation runs to identify the likely vulnerable areas, and then utilizing MCTS search to guide the symbolic execution capable of state forking to move toward those areas.

Algorithm 4: Procedure of Tree Node Simulation in Vital

Input: a tree node *node*, the unsafe pointer set *unsafeSet*
Output: reward of the simulation *reward*

```
1 Function doSimulation(node, unsafeSet):
2     vector<BasicBlock*> simulatedBB
3     simulatedBB = executor.sim_run(node)
4     nu_unsafe = getNumUnsafePt(simulatedBB, unsafeSet)
5     nu_error = executor.numOfMemError()
6     reward = getReward(nu_unsafe, nu_error)
7     return reward
```

When a simulation execution terminates, the reward function is used to calculate the reward for this node. We define the reward function as follows: $F_{reward}(s) = 0.5 * N_{unsafe}(s) + 0.5 * N_{error}(s)$, where the $N_{unsafe}(s)$ denotes the number

of unsafe pointers covered during the simulated run and $N_{error}(s)$ is the memory errors detected throughout the run. We include the detected number of memory errors in the reward function based on a common observation that a root cause of one memory error may lead to various error symptoms in many code regions [72] and code regions related to a buggy code region likely contain bugs too, so we include this number in the reward expecting more memory errors can be found.

Simulation Optimization. When simulating a node in a for-loop statement, the default simulation process repeatedly simulates the same node. Such a process can be inefficient and waste a lot of time. (see more evaluation results in Section 4.3). Therefore, we design a new simulation optimization strategy in VITAL, to reduce the simulation of *unimportant* states that bring little new rewards due to the existence of loops. A straightforward way to avoid an unlimited simulation on a loop is to set the loop bound when doing a simulation. However, this may still lead to unsound results [2]. Our insight is that instead of setting a fixed loop bound for the simulation, we consider a degree of increment based on the rewards of the previous simulation on demand. Therefore, our solution is that when simulating a node in a loop, we record its reward after each simulation run of the loop. If it cannot get a higher reward after a limited number of iterations (we provide an extra option "*–optimization-degree*" to allow users flexible control), it will not repeat the simulation of that node.

3.2.4 Backpropagation. Backpropagation is the final phase in MCTS-guided sampling, during which the total reward and visit count for each state are iteratively updated. For each state, the reward is adjusted based on the simulation outcome, and the visit count is incremented, with updates propagating from the leaf node to the root (performed at Line 11 in Algorithm 1).

3.3 Implementation of VITAL

We implemented VITAL on top of KLEE (v3.0). Following the instructions on the webpage, users can set up and run VITAL to find potential vulnerabilities in the test programs automatically.

Both the implementation of the type inference system and the MCTS algorithm are written in the C++ programming language. For the type inference system used in VITAL, we forked `CCured` (built on top of a static analysis tool SVF [56]) from a previous work [30]. We added a new search strategy `MCTSSearcher` in the `Search` class in KLEE's implementation to support the vulnerability-oriented path exploration. In general, we implemented our own `selectState` and `update` to maintain execution states. The functionality of the MCTS algorithm is implemented in the `selectState` function, where four major functions (i.e., `doSelection`, `doExpansion`, `doSimulation`, and `doBackpropagation`) mentioned in the Algorithm 1 are supported. We also modified the `executeInstruction` and `run` functions in `Executor.cpp` to support the checking of unsafe pointers during execution and the storing of the executed unsafe pointers into a file ("*unsafe-pt.txt*") to help users analyze the program further.

For setting the bias parameter C in Equation in Section 3.2.1 and the simulation optimization option "*– optimization-degree*", we set the value of $\sqrt{2}$ and 700 for them after evaluating their impacts, respectively (more detailed results and discussion are presented in Section 5).

4 Evaluation

Extensive experiments are conducted to evaluate the effectiveness of the proposed VITAL from various perspectives. More specifically, we consider the following four research questions (RQs).

- **RQ1**: Can VITAL cover more unsafe pointer?

- **RQ2**: Can VITAL effectively detect known vulnerabilities?
- **RQ3**: Can each major component contribute to VITAL?
- **RQ4**: Can VITAL detect new (i.e., previously unknown) vulnerability?

Among these RQs, RQ1 evaluates VITAL's capabilities in terms of unsafe pointer coverage and memory error detection compared with representative path exploration strategies in KLEE and CBC. RQ2 aims to investigate the vulnerability detection capabilities in terms of time efficiency and memory usage compared with other approaches over known CVE vulnerabilities. RQ3 concentrates on the contribution of each component of VITAL. RQ4 demonstrates the practical vulnerability detection capability of VITAL.

All experiments conducted in this study ran on a Linux PC with Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz x 12 processors and 64GB RAM running Ubuntu 18.04 operating system.

4.1 RQ1: Unsafe-pointer Covering Capability

Benchmarks. We use the well-known GNU `Coreutils` (v9.5) in this RQ1, following many existing works [9, 28, 35, 61, 64]. The utilities include the basic file, shell, and text manipulation tools of the operating system. We selected 75 utilities for this study and excluded some utilities that: (1) cause nondeterministic behaviors (e.g., `kill`, `ptx`, and `yes`) and (2) exit early due to the unsupported assembly code or the call for external functions.

For the selected utilities, we measure the size of executable lines of code (ELOC) by counting the total number of executable lines in the final executable after global optimization. The distribution of ELOC ranges from 800-8,000, which could comprehensively evaluate the effectiveness of VITAL on test programs of various lengths.

Approaches and Evaluation Metrics for Comparison. We select six representative path exploration strategies as follows, including three commonly evaluated in the literature [35, 62, 64, 65] and three coverage-guided heuristics:

- Breadth first search (`bfs`) and depth first search (`dfs`).
- Random (`random-state`) randomly selects a state to explore.
- Code coverage-guided (`nurs:covnew`) selects a state that has a better chance to cover new code.
- Instruction coverage-guided (`nurs:md2u`) prefers a state with minimum distance to an uncovered instruction, while (`nurs:icnt`) picks a state trying to maximize instruction count.
- CBC [71] proposes compatible branch coverage-driven path exploration for symbolic execution.
- CGS [57] targets on exploring concrete branches that are neglected while performing path exploration.
- FEATMAKER [73] proposes a feature-driven path exploration strategy to improve code coverage.
- EMPC [69] utilizes path cover information for new path prioritization to boost path exploration.

Note that the default implementation of KLEE, CBC, CGS, FEATMAKER, and EMPC does not support the recording of unsafe pointer coverage, and we have modified their source code to enable this functionality for a fair comparison.

We use the following two metrics to assess the effectiveness of different path search strategies:

- **(1) The number of unsafe pointers covered** compares the pointer covering capabilities of various approaches.
- **(2) The number of memory errors detected** compares memory error detection capabilities of various approaches.

Running Setting. Following existing studies [9, 35, 64], we set a running time of one hour for each setting. For random searches, we ran them five times and reported the median results.

Results. Table 1 shows the overall results, where the numbers of two metrics and the improvement achieved by VITAL are recorded. In the table, the column *Total* represents the total number of unsafe pointers covered or memory errors detected by each approach, while *Imp* indicates the improvement (in percentage) achieved by VITAL compared to each approach. $Total_{error}$ represents the total number of memory errors detected by each approach, while Imp_{error} indicates

Table 1. Results compared with prior search strategies

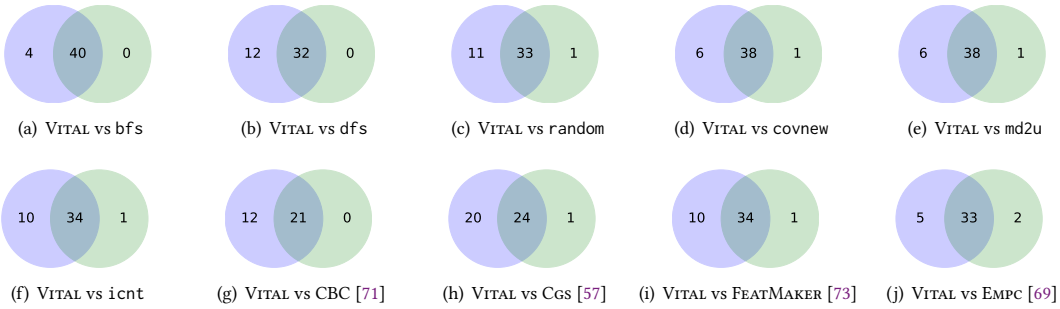
Search Strategies	Unsafe Pointers				Memory Errors	
	Total	Imp_{total}	$Total_{error}$	Imp_{error}	*Total*	Imp_{total}
`bfs`	11610	11.01%	2361	14.95%	40	10.00%
`dfs`	9626	33.89%	1603	69.31%	32	37.50%
`random`	7609	69.38%	1825	48.71%	34	29.41%
`nurs:covnew`	10232	25.96%	2685	3.39%	39	12.82%
`nurs:md2u`	9417	36.86%	1980	37.07%	39	12.82%
`nurs:icnt`	6782	90.03%	1635	66.00%	35	25.71%
CGS [57]	8406	53.53%	1296	109.41%	25	76.00%
FEATMAKER [73]	8185	57.46%	1531	77.27%	34	29.41%
EMPC [69]	6098 (7145)*	17.17%	2096 (2289)*	9.21%	35 (38)*	8.57%
CBC [71]	2194 (3890)*	77.30%	597 (1168)*	95.64%	21 (33)*	57.14%
VITAL	12888	-	2714	-	44	-

* CBC [71] can only successfully run 50 out of 75 utilities, so when comparing to CBC, we only run VITAL over the same 50 utilities as CBC, and the numbers in the parentheses are the numbers of unsafe pointers/memory errors covered/detected by VITAL for the 50 utilities only.

the improvement (in percentage) achieved by VITAL compared to each approach. From the table, it is evident that VITAL performs significantly better than existing search strategies in terms of both the number of covered unsafe pointers and detected memory errors. In particular, VITAL could cover 90.03% more unsafe points compared to `nurs:icnt` and detect 57.14% more memory errors compared to CBC, demonstrating the superior performance of VITAL. Compared to CGS, FEATMAKER, and EMPC, VITAL also outperforms them in both metrics. Specifically, VITAL covers 53.53%, 57.46%, and 17.17% more unsafe pointers in total and detects 109.41%, 77.27%, and 9.21% more memory errors than CGS, FEATMAKER, and EMPC, respectively. Note that following a prior work [35], we did not submit these reports to developers, as we target evaluating detection accuracy, not responsible disclosure. This choice is justified by the fact that Vital builds on KLEE—a tool widely adopted in research and industry and known to produce true positives—so we follow KLEE's assumption that all reported errors (all out-of-bounds issues) are true positives.

We also investigate the *unique* memory error detection capability of VITAL, and the Venn Figure 4 presents the overall results. From the figure, we can observe that VITAL could detect larger numbers (ranging from 4 compared to `bfs` and 20 to CGS [57]) of unique memory errors. In particular, compared to the latest searching strategies in the last four Venn plots, VITAL covers almost all memory errors detected by the baselines while contributing many additional ones. Compared to CBC, VITAL contributes 12 unique detections and overlaps on 21, with CBC adding none. Compared to CGS, VITAL adds 20 unique overlaps on 24, and CGS contributes only 1 unique case. Against FEATMAKER, VITAL adds 10 unique with 34 overlaps and just 1 unique for FEATMAKER. Against EMPC, VITAL contributes 5 unique with 33 overlaps, and EMPC adds 2 unique. Overall, VITAL captures 95–100% of the combined detections in each pair and consistently provides the largest set of exclusive findings, indicating a better memory error detection capability. VITAL misses only a few memory errors detected by other strategies (up to 2 when compared to EMPC), this is mainly because the MCTS algorithm may be biased toward more promising or frequently visited branches, which is a well-known issue in MCTS design [6], potentially overlooking branches that contain critical but infrequent behaviors.

Compared to the recent CBC search strategy, VITAL outperforms it for two reasons. First and most importantly, the assumption that many paths have no contribution to branch coverage and thus bug finding held by CBC is not always valid for memory error detection, as most of the memory errors can only be triggered under specific contexts (e.g., function calls). The vulnerable function may trigger the bug in one function call, but invoking another function call may not. Second, computing dependence information on demand in CBC takes time and memory consumption (as



(a) VITAL vs bfs (b) VITAL vs dfs (c) VITAL vs random (d) VITAL vs covnew (e) VITAL vs md2u

(f) VITAL vs icnt (g) VITAL vs CBC [71] (h) VITAL vs CGS [57] (i) VITAL vs FEATMAKER [73] (j) VITAL vs EMPC [69]

Fig. 4. Number of detected *unique* memory errors by VITAL compared to other search strategies

reported in the CBC paper [71], it introduces 30% more memory consumption and execution time than the baseline approach KLEE when performing path exploration). Compared to CGS, FEATMAKER, and EMPC, VITAL also detects most of the unique memory errors, indicating its superior performance. This is mainly because these approaches all aim to improve code coverage (the concrete branches explored by CGS; the specific features targeted by FEATMAKER; and the path cover information utilized by EMPC) while being agnostic to vulnerable behaviors, making them less effective at detecting unique memory errors. Although it may be possible to modify them to be vulnerability-oriented, it can be hard to perform a smart path search that considers both forward and backward program information during path exploration, as does VITAL (see more detailed comparison and discussion in Section 6). VITAL, on the other hand, directly targets exploring unsafe program paths, thus achieving better performance in both metrics.

Answer to RQ1: Given a one-hour execution period, Vital surpasses current coverage-guided path search approaches by covering up to 90.03% of unsafe pointers and uncovering up to 57.14% more unique memory errors.

TAKEAWAY: We analyze the correlation between the number of unsafe pointers and memory errors and present the result in Figure 1 using the data collected in Table 1. The statistical Pearson's coefficient **0.924** suggests a strong positive correlation [16], meaning a higher number of unsafe pointers suggests a greater likelihood of vulnerabilities.

4.2 RQ2: Vulnerability Detection Capability

Benchmarks. We use four CVEs in GNU `libtasn1` library with different versions (followed Chopper [61]), namely CVE-2012-1569 (v3.21), CVE-2014-3467 (v3.5), CVE-2015-2806 (v4.3), and CVE-2015-3622 (v4.4). The `libtasn1` library facilitates the serialization and deserialization of data using Abstract Syntax Notation One (ASN.1). The selected vulnerabilities predominantly involve memory out-of-bounds accesses. Note that each identified vulnerability is associated with detecting a single failure, except for CVE-2014-3467, where this vulnerability manifests across three distinct locations, so the experiment seeks to detect six distinct vulnerabilities.

Approaches and Evaluation Metrics for Comparison. We compared VITAL with the baseline approach (i.e., KLEE [9]) and five state-of-the-art approaches (i.e., `Chopper` [61], CBC [71], CGS [57], FEATMAKER [73], and EMPC [69]) in this RQ. To be specific, following existing studies [61, 71], we run KLEE, `Chopper`, and CBC under different search strategies, i.e., *Random* (`random`), *DFS* (`dfs`), and *Coverage* (`nurs:covnew`), while keep other with default search strategy. We use the following two metrics to assess their effectiveness.

- **(1) Execution time** records the time to detect a vulnerability.
- **(2) Memory consumption** measures memory usage in detecting a vulnerability.

Table 2. Results of time on detecting known CVE vulnerabilities.

Approaches	Search	2012-1569	2014-3467[1]	2014-3467[2]	2014-3467[3]	2015-2806	2015-3622
KLEE [9]	random	00:11:36	*Time-out*	00:00:06	*Time-out*	00:05:43	*Time-out*
	dfs	00:02:40	00:03:07	14:55:21	*Time-out*	02:55:37	*Time-out*
	coverage	00:11:03	OOM	00:00:05	*Time-out*	OOM	07:49:39
Chopper [61]	random	00:01:50	00:08:24	00:09:25	00:26:48	00:02:33	00:00:58
	dfs	00:01:03	00:00:12	00:58:48	00:00:29	*Time-out*	00:13:56
	coverage	00:01:57	00:04:22	00:19:11	00:11:22	00:01:48	00:00:57
CBC [71]	random	00:01:01	*N/A*	00:00:29	00:00:54	*N/A*	00:02:40
	dfs	00:00:48	*N/A*	*Time-out*	00:00:18	00:21:48	00:01:40
	coverage	00:01:18	*N/A*	00:00:23	*Time-out*	*N/A*	00:01:19
CGS [57]	cgs	00:11:18	18:18:09	00:00:14	00:33:40	00:50:29	00:10:01
FEATMAKER [73]	featmaker	00:27:23	*Time-out*	00:00:11	00:49:09	00:01:24	00:32:30
EMPC [69]	empc	-	OOM	00:00:10	*Time-out*	00:17:37	00:00:38
VITAL	mcts	00:01:03	00:01:06	00:00:19	00:00:09	00:02:53	00:00:26

* The time format is *hour:minute:second*. "*N/A*" represents the normal termination of execution without reproducing the vulnerability; "*OOM*" means Out-of-Memory. *Time-out* means the execution exceeded the time limit without reproducing the vulnerability. The index number n in CVE-2014-3467^n represents a distinct location of the vulnerability manifested. The '-' in the row of EMPC indicates that EMPC cannot successfully reproduce the CVE.

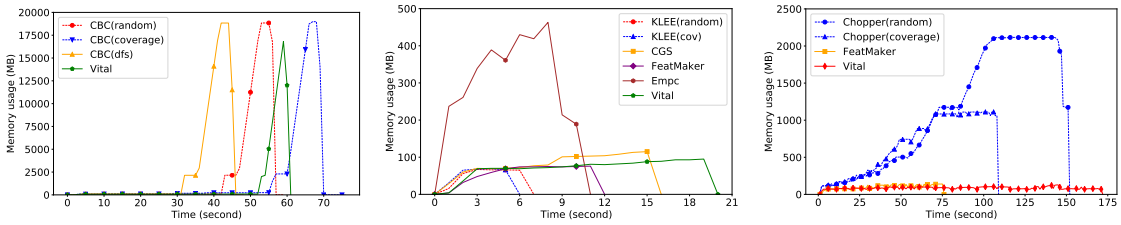
Running Setting. Following the existing study [61], we set a timeout of 24 hours to detect each vulnerability. Again, for random searches, we ran them five times and reported the median results.

Results. Table 2 presents the overall results in terms of execution time. From the table, we can see that VITAL outperforms Chopper and CBC for almost all vulnerabilities. Upon detecting the vulnerability in CVE-2014-3467[2], VITAL achieves a speedup of 30x to Chopper. For the vulnerability CVE-2015-2806, VITAL takes slightly more time (less than 20 seconds) to detect it. This is because Chopper skips some large functions before execution, making it detect the vulnerability faster. However, as emphasized in Section 1, users of Chopper need prior expert knowledge to decide which functions/lines to skip, which could be a labor-intensive and time-consuming task. Also, Chopper consumes more memory when detecting the vulnerability (see more details below).

Compared to CBC[3], VITAL takes less time to detect 5 out of 6 vulnerabilities (except CVE-2012-1569 with *Random* and *DFS* search). For CVE-2014-3467[1], CBC does not detect it in any search strategies as this vulnerability happens in a while-loop condition, which requires multiple executions of the condition to trigger the vulnerability. This again emphasizes a fundamental deficiency in CBC, as it can only detect bugs about assertion failure that are triggered when the code executes once. However, as shown in previous studies [1, 11, 27], a plethora of memory errors occur in loop iterations, and only executing the loop once is insufficient to detect the important category of memory errors. Compared to the remaining CGS, FEATMAKER, and EMPC, VITAL is still superior, as VITAL detects all the vulnerabilities while others miss some of them. For those cases that others can detect, VITAL also takes less time to detect them in 4 out of 6 cases.

In terms of memory consumption, we run VITAL against Chopper and CBC over two vulnerabilities in CVE-2015-2806 and CVE-2012-1569, respectively, to gain more insights because VITAL takes comparable or slightly more time for the vulnerability detection. Figure 5 shows the results, where *x-axis* represents the time to detect the vulnerability and *y-axis* indicates the usage of memory. From the two figures, we can observe that VITAL consistently consumes less memory when detecting the vulnerability. In particular, as shown in Figure 5(a), we can observe that VITAL consumes significantly less memory when detecting the same vulnerabilities compared to Chopper. For example, Chopper with a random search consumes at most 2,115 MB of memory, while VITAL only takes around 100 MB of memory, producing a

[3]There are some differences between the results in Table 2 and those in [71] Table 2, mainly due to different settings used. We use a *fixed* setting that produces the best overall results, instead of the *best setting for each* CVE, which requires expert knowledge about each CVE, and we think would not be a practical usage (details in Appendix-A in the Supplementary Material or online at Appendix).



(a) CVE-2012-1569 (b) CVE-2014-3457[2] (c) CVE-2015-2806

Fig. 5. Results of memory consumption (note that only VITAL takes comparable or slightly more time are presented)

significant reduction (i.e., 20x) in memory consumption. This is reasonable as `Chopper` takes a state recovery mechanism to maintain the execution of skipped functions. Since the skipped functions can be very large, maintaining recovered states in `Chopper` tends to be memory-intensive. Compared to CBC, it requires extra memory consumption to obtain the dependence information during the symbolic execution, which is memory-intensive. In contrast, VITAL does not increase memory consumption compared to standard symbolic execution, providing a solution that is not only lightweight but also highly effective. Compared to EMPC over CVE-2014-3467[2], although VITAL takes slightly more time than EMPC, VITAL consumes 5x less memory usage, demonstrating its efficiency.

The superior performance from Vital is expected, as existing approaches all aim to improve code coverage while being agnostic to vulnerable behaviors, making them less effective at detecting vulnerabilities. Specifically, CGS focuses on exploring concrete branches, which may not necessarily lead to the exploration of paths that trigger vulnerabilities. FEATMAKER targets specific features, which may not align with the paths that expose vulnerabilities. EMPC utilizes path cover information, which may not prioritize paths that lead to vulnerabilities. In contrast, VITAL directly targets exploring unsafe program paths, thus achieving better performance.

Answer to RQ2: VITAL outperforms chopped symbolic execution by achieving a speedup of up to 30x execution time and a reduction of up to 20x memory consumption.

4.3 RQ3: Ablation Studies

Benchmarks and Evaluation Metrics. We use the same benchmarks and evaluation metrics as in RQ1 to compare different variant approaches of VITAL to answer RQ3 in this subsection.

Variant Approaches. We design several variants of VITAL to gain a deeper understanding of the contribution of each component. We focus on evaluating the following variant approaches:

- VITAL(¬EXP) uses random expansion without guided expansion.
- VITAL(¬SIM) performs path search without simulation.
- VITAL(¬SOPT) simulates without the optimization for loops.
- VITAL(OPT2-x), where x represent the number of instructions to terminate the simulation of *OP2* (as discussed in Section 3.2.3, *OP2* terminates the simulation when x instructions are executed).

Running Setting: We use the same setting as in RQ1: each approach with a one-hour run and use the metrics (i.e., the number of unsafe pointer coverage and memory errors detected) to evaluate their effectiveness.

Results. Table 3 presents the overall results. We can conclude that VITAL performs better than all variant approaches. Specifically, compared to VITAL(¬EXP), VITAL could cover 16.80% more unsafe points and detect 7.32% more memory errors, demonstrating the contribution of unsafe pointer-guided node expansion designed in Algorithm 1. The superior

Table 3. Comparison results with variant approaches

Search	Unsafe Pointers		Memory Errors	
	Num_{total}	Imp_{total}	Num_{total}	Imp_{total}
VITAL(¬EXP)	11034	16.80%	41	7.32%
VITAL(¬SIM)	9010	43.04%	36	22.22%
VITAL(¬SOPT)	9863	30.67%	33	33.33%
VITAL(OPT2-500)	5449	137.52%	25	76.00%
VITAL(OPT2-1000)	7805	65.12%	26	69.23%
VITAL(OPT2-2000)	6477	98.98%	26	69.23%
VITAL	12888	-	44	-

performance of VITAL is reasonable as random expansion tends to waste exploration efforts on non-vulnerable paths, thus downgrading the overall performance of MCTS. The other two variant approaches share similar results, where VITAL achieves an improvement of 43.04% and 22.22% as well as 30.67% and 33.33% in terms of the number of covered unsafe pointers and detected memory errors, compared to VITAL(¬SIM) and VITAL(¬SOPT), respectively, indicating the contribution of simulation and its optimization. The results of VITAL(¬SIM) is justified that without simulation, MCTS cannot fully utilize the past execution information to guide the path exploration, thus downgrading its performance. The results of VITAL(¬SOPT) also demonstrate the effectiveness of the loop optimization in improving the simulation performance, as loops are widely used in real-world programs.

Evaluation of Different Simulation Strategies. We run *OP3* (the default simulation setting in VITAL), VITAL(OPT2-500), VITAL(OPT2-1000), and VITAL(OPT2-2000) (where x in VITAL(OPT2-x) means the fixed number of instructions to terminate the simulation) to evaluate different settings of *OP2*. We omit *OP1* as its static simulation of a node tends to be ineffective due to time-consuming CFG traversing and high false alarm rate, as discussed in Section 3.2.3. As shown in Table 3, all three settings of *OP2* perform inferior compared to *OP3* (normally terminated execution) in VITAL. This is reasonable as aborting too early would make VITAL prone to false negatives, thus missing interesting unsafe pointers coverage and downgrading the overall performance of MCTS.

Comparison with Unsafe Pointers Guided Search without MCTS. Another simpler variant approach is to use the indicator (i.e., type-unsafe pointers) to guide the search without using MCTS. Theoretically, the simpler heuristic is limited for two reasons. First, like many existing search techniques (as thoroughly discussed in Table 4 and in Section 6), the simpler heuristic is agnostic to the past execution, which is shown to be an important factor to boost the path exploration. Second, the limited capability of balancing the exploration of future unexplored paths and the exploitation of the past explored paths during path search makes it costly and difficult to reach the known location of type-unsafe pointers. In contrast, Vital is superior for both aspects: MCTS can utilize the past execution information to guide the path exploration, and it has a good potential to balance the exploration and exploitation during path search, making Vital an optimal solution towards the vulnerability-guided path exploration.

Answer to RQ3: The newly designed components, including type-unsafe pointer-guided node expansion and node simulation as well as its loop optimization, are imperative to boost the performance of VITAL in terms of covering unsafe pointers and detecting memory errors.

4.4 RQ4: Practical Application of VITAL

To demonstrate the practical vulnerability detection capability of VITAL over large-scale software systems, we run VITAL over the intensively-tested and latest released *objdump* package (includes more than 20k lines of code) in GNU

`binutils-2.44`. As a result, VITAL detected a previously unknown vulnerability[4] within one minute. The issue is a memory leak where an allocated object is not appropriately de-allocated, which may cause severe security risks such as critical information leakage or denial of service. This issue had been lurking for more than 7 years before we reported it[5]. Since the critical impact of the issue, developers confirmed and fixed it swiftly within 8 hours after we submitted the bug report. It has been assigned a new CVE ID (i.e., CVE-2025-3198).

It is worth noting that other approaches face difficulties in detecting this vulnerability. To have a better understanding, we ran KLEE, `Chopper`, and CBC on the same version of *objdump* for 24 hours but it turns out that none of them could detect this issue[6]. In summary, KLEE failed to cover the vulnerable path because of the complex input conditions and multiple loop iterations required to reach the vulnerable function. `Chopper` [61] failed to detect the issue mainly due to the lack of prior expert knowledge or technical support to set up the execution. KLEE's default search heuristics are limited by bypassing the input-dependent loops, which are prerequisites to reach the target vulnerable function. CBC [71] is restricted by the non-vulnerability-oriented path search heuristic, which ignores the fact that many memory errors can only happen after multiple executions on certain loops [27].

VITAL covers the vulnerable path because of its new way of simulating execution paths and its novel search based on MCTS. Compared with existing solutions, VITAL can skip *unimportant* loops that have no contribution to the accumulation of unsafe pointer coverage by using simulation optimization strategies (see Section 3.2.3). For loops that may lead to memory issues, our simulation and MCTS will evaluate the reward of each of their iterations and continue when new unsafe pointers are covered.

Answer to RQ4: VITAL is able to detect previously unknown vulnerabilities in practice, working in a fully automatic manner without the need for prior expert knowledge.

5 Discussion

Overhead of Pointer Type Inference. To further understand the overhead of the type inference system (i.e., `CCured`) used in VITAL, we measure the time to infer pointer types. The results show that in most cases (64%, 48 out of 75), the time spent on the analysis is within 5 seconds. We believe that such overheads are negligible compared to the large amount of time used for the entire testing period (e.g., 24 hours in RQ2).

Impact of Different Configurations. The selection of the value of the parameter `C` defined in Equation in Section 3.2.1 and "*–optimization-degree*" in Algorithm 4 may affect the performance of VITAL. Thus, we conduct extra experiments to assess the impact of different running configurations.

For the parameter `C` in Section 3.2.1, we run VITAL with values of $\sqrt{2}$, 5, 10, 20, 50, and 100. The results show that VITAL covered 12,888, 12,620, 13,144, 12,724, 13,188, and 12,578 unsafe pointers and detected 44, 41, 42, 43, 43, and 44 memory errors in each configuration, respectively. Since the goal of VITAL is to detect more memory vulnerabilities, we select the value $\sqrt{2}$ as the default configuration of VITAL as this setting yields the best memory error detection and comparable unsafe pointer coverage capabilities.

For the impact of "*–optimization-degree*", we run VITAL with values of 100, 300, 500, 700, 1,100, and 1,500. The results show VITAL covered 12,078, 12,349, 12,660, 12,888, 12,655, and 12,578 unsafe pointers and detected 35, 39, 42, 44, 41, and

[4] https://sourceware.org/bugzilla/show_bug.cgi?id=32716.
[5] The oldest version that can reproduce the issue is binutils-2.29 released on 25/09/2017.
[6] We provide detailed vulnerability analysis in Appendix-B in the Supplementary Material or online at Appendix.

39 memory errors under each configuration, respectively. Since the value of 700 produces the best results in terms of both unsafe pointer coverage and memory error detection capabilities, we set 700 as the default value in VITAL.

Threats to Validity. The *internal* validity concerns stem from the implementation of VITAL. To mitigate this threat, we built VITAL on top of the well-maintained and recently released version (v3.0) of KLEE [9]. In addition, we have meticulously implemented VITAL, reusing existing APIs in KLEE, as explained in Section 3.3, and have performed a detailed code check to mitigate the threat.

The *external* threat comes from the benchmarks used in this study. We used GNU `Coreutils` and a library `libtasn1` with four different versions. Although they have been widely used for evaluating symbolic execution [9, 28, 35, 60, 62], these programs may not be representative enough for various software systems. To further alleviate these potential threats, we are committed to expanding the program sets in our future work. To mitigate such a threat, we also evaluated VITAL on a larger test program in RQ4 and detected an unknown vulnerability.

The *construct* validity threat is subject to configurations of parameters. We address this concern by thoroughly investigating their impact, which enhances transparency and enables a deeper understanding of the influence of different configurations. Following many existing studies [20, 25, 26, 32, 42], another threat of VITAL is that the current version supports ensuring spatial memory safety only. This is mainly because the tool used for type inference `CCured` [43] can not classify all unsafe pointers (i.e., the ones that lead to *temporal* memory errors). Note that extending the support to more types of *new* indicators to detect more types of vulnerabilities in VITAL should only involve engineering effort (e.g., transporting the tool SAFECode [21] to work on compatible LLVM bitcode to collect *unsafe* pointers to help detect temporal memory errors). We plan to alleviate this threat in future work.

Limitations. VITAL suffers from certain inherent limitations in symbolic execution engines (e.g., KLEE [9]) due to limited memory modeling, environment modeling, and efficiency issues, which may restrict the memory error detection capability of VITAL. This is because some intractable vulnerabilities can only be triggered under a complex situation, which requires a more comprehensive modeling of the program semantics of test programs. Efficiency is also an issue, as most symbolic execution engines analyze test programs by interpreting the intermediate representation code (e.g., LLVM Bitcode in KLEE), which is shown to be inefficient [45, 46]. Recent studies [44, 51, 64, 65] have been proposed to resolve these problems, and we plan to integrate them into VITAL in future work.

Following many studies [20, 25, 26, 32, 42], another limitation of VITAL, which we inherit from the static analysis tool `CCured` [43], is that the current VITAL ensures *spatial* memory safety only. This is mainly because `CCured`, the tool used for type inference, can not classify all unsafe pointers (e.g., the ones leading to *temporal* memory errors). There are several directions to address this limitation. First, one may add the indicators of temporal memory errors (e.g., dangling pointers) to guide the path exploration in VITAL. Note that extending the support to more types of *new* indicators to detect more types of vulnerabilities in VITAL should only involve engineering effort (e.g., transporting the SAFECode tool [21] to work on compatible LLVM bitcode to collect *unsafe* points to help detect temporal memory errors). We plan to leverage more advanced techniques (such as those proposed in DataGuard [30]) to address this limitation in future work. Second, one may employ advanced static analysis tools such as Infer [19], which offers comprehensive detection of memory safety issues (including temporal memory), to guide path exploration. However, unlike `CCured`, which has the guarantee of soundness (i.e., free of false negatives) by design, Infer and other tools may suffer from both false positives and negatives [18, 37]. To enable a more robust cooperation with other static analysis tools beyond `CCured`, one has to design an appropriate solution to reduce false positive and negative rates before using their results to guide the path exploration for symbolic execution. We consider such a direction for future work as well.

6 Related Work

Techniques for Alleviating Path Explosion. Various techniques are introduced to tackle the path explosion problem, the most related include path search strategies and under-constrained (e.g., *chopped*) symbolic execution. Most search heuristics for symbolic execution are coverage-guided. Cadar *et al.* [10] propose a Best-First Search strategy. Cadar *et al.* further propose KLEE [9], where random and code coverage-guided (`nurs:covnew`) search strategies are proposed. Later, KLEE continued to upgrade to support many more strategies, such as `bfs`, `dfs`, and instruction coverage-guided (`nurs:md2u` and `nusrs:icnt`). Burnim *et al.* [7] propose using a weighted control flow graph (CFG) to guide exploration to the nearest uncovered parts based on the distance in the CFG. Li *et al.* [35] propose to exploit a new *length-n* subpath program spectra to systematically approximate full path information for guiding path exploration. He *et al.* [28] adopt a machine learning-based strategy to first train a model based on program execution information and then effectively select promising states with the trained model. UBSYM [3] exploits test units that are relevant to vulnerability to prioritizes the path exploration in binary programs. CBC [71] proposes a compatibility-based search strategy to prioritize the execution states that are more likely to be compatible with other states, thus increasing the chance of generating new paths. CGS [57] proposes a concrete-path search strategy to prioritize the execution states that can lead to new paths through concrete execution. FEATMAKER [73] proposes a feature-based (i.e., path conditions) search strategy to prioritize the execution states that can maximize the performance of symbolic execution in terms of code coverage and bug detection. EMPC [69] proposes a new path exploration strategy based on minimum path cover (MPC), where MPC provides an option for runtime path selection to use the least number of paths to maximize code coverage. Unlike conducting a path search only for code coverage, there are only a few vulnerability-oriented search strategies. `StatSym` [68] instruments test programs to construct predicates that indicate vulnerable features and then employs a path construction algorithm to select the vulnerable paths. `SyML` [50] guides path exploration toward vulnerable states through pattern learning. However, existing coverage-guided techniques give the same priority to code that is unlikely to contain vulnerabilities. In contrast, VITAL maximizes the number of unsafe pointers to increase the likelihood of exposing memory unsafe vulnerabilities. Furthermore, vulnerability-oriented approaches require a training set of vulnerabilities previously discovered in the program and try to link patterns in the runtime information to vulnerabilities. In contrast, VITAL requires neither training nor unspecific runtime information to quantify the vulnerability-proneness of a path. Since both `StatSym` and `SyML` are not open-sourced and `SyML` only focuses on binary programs, we could not directly experimentally compare VITAL with them in this paper.

Regarding under-constrained symbolic execution, Csallner *et al.* [17] and Engler *et al.* [23] propose and extend the idea of under-constrained symbolic execution, where the symbolic executor cuts the code (e.g., an interesting function) to be analyzed out of its enclosing system and checks it in an isolation manner. Recent work `Chopper` [61] cuts out uninteresting functions that are vulnerability irrelevant.

To help readers better understand the differences between VITAL and existing path exploration techniques, we summarize the main differences between the techniques compared in the evaluation and VITAL in Table 4. We compare them from four perspectives: (1) whether the technique is vulnerability-oriented; (2) whether the technique is fully automated without requiring expert knowledge; (3) whether the technique can learn from past execution information to improve path exploration, and (4) whether the technique can balance exploration of *unexplored* paths and exploitation of *explored* paths during path search. As shown in Table 4, VITAL is the only technique that supports all four features to improve path exploration. Specifically, compared with existing representative coverage-guided or other heuristics for path exploration (i.e., KLEE [9], CBC [71], CGS [57], FEATMAKER [73], and EMPC [69]), VITAL is the only one that

Table 4. Comparison of different path explorations in symbolic execution (✔ represents holds the feature while ✘ does not).

Features	KLEE [9]	Chopper [61]	CBC [71]	CGS [57]	FEATMAKER [73]	EMPC [69]	VITAL
Vulnerability-oriented?	✘	✔	✘	✘	✘	✘	✔
Fully automated?	✔	✘	✔	✔	✔	✔	✔
Learn from past execution?	✘	✘	✘	✘	✘	✘	✔
Balance exploration&exploitation?	✘	✘	✘	✘	✘	✘	✔

is vulnerability-oriented, can learn from past execution information, and can balance exploration and exploitation to improve path exploration. Furthermore, compared with under-constrained symbolic execution, `Chopper` requires users to specify which functions to skip based on their expert knowledge and cannot learn from past executions. In contrast, VITAL is fully automated without requiring expert knowledge. In summary, compared to existing path search strategies, VITAL is a novel technique that can automatically perform vulnerability-oriented path exploration using a novel MCTS-based approach that not only learns from past execution but also balances exploration and exploitation during path search, making it advanced in vulnerability-oriented path exploration in symbolic execution.

Applications of Monte Carlo Tree Search. MCTS was initially applied to enhance heuristics for a theorem prover [24]. Then, it was used to optimize program synthesis [36] and symbolic regression [67]. Furthermore, MCTS is employed in Java PathFinder [47], where it was used for explicit state model checking. Liu *et al.* [38] adopt MCTS to achieve the best trade-off between concolic execution and fuzzing for coverage-based testing. Zhao *et al.* [77] model the seed scheduling problem in fuzzing as a decision-making problem and use MCTS to select optimal seeds. The works most related to ours are the `canopy` [40] and the approach proposed by Yeh *et al.* [70]. `canopy` uses MCTS to guide the search for costly paths, where the cost is defined as memory consumed and execution time along a path. Yeh *et al.*' approach utilizes MCTS to select valuable paths, where the valuable refers to the number of visited basic blocks.

VITAL adopts MCTS for vulnerability-oriented path exploration for symbolic execution. Compared with the most related works, the differences are as follows. (1) Our goal is to detect vulnerable paths in C/C++ programs, while `canopy` aims to find the costly paths in Java programs, and Yeh *et al.*' approach focuses on the path with the largest executed number of basic blocks in binary programs. (2) The node expansion designed in VITAL is guided by the results of static program analysis, that is, type inference, while both the two compared approaches apply a random expansion strategy. (3) The simulation policy designed by VITAL is optimized by previous simulation outcomes, while `canopy` adopts random simulation with limited optimizations and Yeh *et al.*' approach uses CFG of the binary program to perform simulation (which may yield imprecise reward, as recovering CFG from binary programs is an undecidable problem [52]). In summary, VITAL tends to be more applicable by design to detect new vulnerabilities in practice.

7 Conclusion with Future Work

We present VITAL, a new vulnerability-oriented symbolic execution via type-unsafe pointer-guided Monte Carlo Tree Search. VITAL guides the path search toward vulnerabilities by (1) acquiring *type-unsafe* pointers by a static pointer analysis (i.e., type inference), and (2) navigating an optimal exploration-exploitation trade-off to prioritize program paths where the number of unsafe pointers is maximized, leveraging unsafe pointer-guided Monte Carlo Tree Search. We have compared VITAL with existing path search strategies and chopped symbolic execution, and the results demonstrate the superior performance of VITAL among existing approaches in terms of unsafe pointer coverage and memory errors/vulnerabilities detection capability, specifically for practical vulnerability detection capability. For future work, we are actively pursuing to strengthen vulnerable behavior analysis and modeling in VITAL.

Acknowledgments

We appreciate Cristian Cadar, Martin Nowack, and Daniel Schemmel for their constructive insights in the earlier stages of this project. We also thank anonymous reviewers who provided their insightful comments on the previous versions of the draft and the developers who helped promptly confirm and fix the bug we reported.

References

[1] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic Exploit Generation. *Communication of ACM* 57, 2 (2014), 74–84.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Computing Survey* 51, 3, Article 50 (2018), 39 pages.

[3] Sara Baradaran, Mahdi Heidari, Ali Kamali, and Maryam Mouzarani. 2023. A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes. *Int. J. Inf. Secur.* 22, 5 (May 2023), 1277–1290. doi:10.1007/s10207-023-00691-1

[4] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. 30–40.

[5] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 1621–1633.

[6] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 1–43.

[7] Jacob Burnim and Koushik Sen. 2008. Heuristics for scalable dynamic test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 443–446.

[8] Project Zero by Google. 2024. *0-day vulnerabilities In the Wild.* https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/edit?gid=0#gid=0

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 209–224.

[10] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 1–38.

[11] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. 380–394.

[12] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. 559–572.

[13] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *IEEE Symposium on Security and Privacy (S&P)*. 1580–1596.

[14] Ferdinando Cicalese, Balázs Keszegh, Bernard Lidický, Dömötör Pálvölgyi, and Tomáš Valla. 2016. On the tree search problem with non-uniform costs. *Theoretical Computer Science* 647 (2016), 22–32.

[15] Catalin Cimpanu. 2024. *Microsoft: 70 percent of all security bugs are memory safety issues.* https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

[16] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise Reduction in Speech Processing* (2009), 1–4.

[17] Christoph Csallner and Yannis Smaragdakis. 2005. Check'n'Crash: Combining static checking and testing. In *Proceedings of the International Conference on Software Engineering*. 422–431.

[18] Han Cui, Menglei Xie, Ting Su, Chengyu Zhang, and Shin Hwei Tan. 2024. An Empirical Study of False Negatives and Positives of Static Code Analyzers From the Perspective of Historical Issues. *arXiv preprint arXiv:2408.13855* (2024).

[19] Facebook Developers. 2025. *A Static Analyzer for Java, C, C++, and Objective-C.* https://fbinfer.com/docs/all-categories

[20] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. 103–114.

[21] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, New York, NY, USA, 144–157. doi:10.1145/1133981.1133999

[22] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*. 53–60.

[23] Dawson Engler and Daniel Dunbar. 2007. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 1–4.

[24] Wolfgang Ertel, Johann M Ph Schumann, and Christian B Suttner. 1989. Learning heuristics for a theorem prover using back propagation. In *Proceedings of Österreichische Artificial-Intelligence-Tagung*. 87–95.

[25] Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee. 2023. TAILCHECK: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 535–552.

[26] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. 2024. Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags . In *IEEE Symposium on Security and Privacy (SP)*. 4239–4257.

[27] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for {Overflows}: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*. 49–64.

[28] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to Explore Paths for Symbolic Execution. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2526–2540.

[29] Manuel Heusner, Thomas Keller, and Malte Helmert. 2017. Understanding the search behaviour of greedy best-first search. In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 8. 47–55.

[30] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. 2022. The Taming of the Stack: Isolating Stack Data from Memory Errors. In *Proceedings of 29th Annual Network and Distributed System Security Symposium (NDSS)*. 1–17.

[31] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. 2019. Computing Summaries of String Loops in C for Better Testing and Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 874–888.

[32] Piyus Kedia, Rahul Purandare, Udit Agarwal, and Rishabh. 2023. CGuard: Scalable and Precise Object Bounds Protection for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1307–1318.

[33] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European Conference on Machine Learning*. 282–293.

[34] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. 193–204.

[35] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering symbolic execution to less traveled paths. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. 19–32.

[36] Jinsuk Lim and Shin Yoo. 2016. Field report: Applying monte carlo tree search for program synthesis. In *Proceedings of 8th International Symposium Search Based Software Engineering (SSBSE)*. 304–310.

[37] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 544–555.

[38] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin IP Rubinstein. 2020. Legion: Best-first concolic testing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 54–65.

[39] Edward S Lowry and Cleburne W Medlock. 1969. Object code optimization. *Commun. ACM* 12, 1 (1969), 13–22.

[40] Kasper Luckow, Corina S Păsăreanu, and Willem Visser. 2018. Monte Carlo tree search for finding costly paths in programs. In *Proceedings of 16th International Conference on Software Engineering and Formal Methods (SEFM)*. 123–138.

[41] Daniele Midi, Mathias Payer, and Elisa Bertino. 2017. Memory Safety for Embedded Devices with nesCheck. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*. 127–139.

[42] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 245–258.

[43] George C Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 128–139.

[44] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. 2019. Deferred concretization in symbolic execution via fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 228–238.

[45] Pansilu Pitigalaarachchi, Xuhua Ding, Haiqing Qiu, Haoxin Tu, Jiaqi Hong, and Lingxiao Jiang. 2023. KRover: A Symbolic Execution Engine for Dynamic Kernel Analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2009–2023.

[46] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *Proceedings of 29th USENIX Security Symposium (USENIX)*. 181–198.

[47] Simon Poulding and Robert Feldt. 2015. Heuristic model checking using a Monte-Carlo tree search algorithm. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO)*. 1359–1366.

[48] Marco Prandini and Marco Ramilli. 2012. Return-oriented programming. *IEEE Security & Privacy* 10, 6 (2012), 84–87.

[49] Google Chromium Project. 2024. *Memory Safety in Chromium Project.* https://www.chromium.org/Home/chromium-security/memory-safety/

[50] Nicola Ruaro, Kyle Zeng, Lukas Dresel, Mario Polino, Tiffany Bao, Andrea Continella, Stefano Zanero, Christopher Kruegel, and Giovanni Vigna. 2021. SyML: Guiding symbolic execution toward vulnerable states through pattern learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 456–468.

[51] Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar. 2023. KDAlloc: The KLEE Deterministic Allocator: Deterministic Memory Allocation during Symbolic Execution and Test Case Replay. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1491–1494.

[52] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of IEEE Symposium on Security*

and Privacy (S&P). 138–157.

[53] Jeffrey Vander Stoep. 2024. *Memory Safe Languages in Android 13.* https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

[54] STP. 2024. *Simple Theorem Prover, an efficient SMT solver for bitvectors.* https://github.com/stp/stp

[55] Nathan Sturtevant and Ariel Felner. 2018. A brief history and recent achievements in bidirectional search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[56] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*. 265–266.

[57] Yue Sun, Guowei Yang, Shichao Lv, Zhi Li, and Limin Sun. 2024. Concrete Constraint Guided Symbolic Execution. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 122, 12 pages. doi:10.1145/3597503.3639078

[58] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. 2023. Monte Carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review* 56, 3 (2023), 2497–2562.

[59] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)*. 48–62.

[60] David Trabish, Shachar Itzhaky, and Noam Rinetzky. 2021. A bounded symbolic-size model for symbolic execution. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1190–1201.

[61] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 350–360.

[62] David Trabish and Noam Rinetzky. 2020. Relocatable addressing model for symbolic execution. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 51–62.

[63] Haoxin Tu. 2025. *Replication Package of Vital.* https://github.com/haoxintu/Vital-SE

[64] Haoxin Tu, Lingxiao Jiang, Xuhua Ding, and He Jiang. 2022. FastKLEE: faster symbolic execution via reducing redundant bound checking of type-safe pointers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1741–1745.

[65] Haoxin Tu, Lingxiao Jiang, Jiaqi Hong, Xuhua Ding, and He Jiang. 2024. Concretely Mapped Symbolic Memory Locations for Memory Error Detection. *IEEE Transactions on Software Engineering* 01, 01 (2024), 1–21.

[66] David A Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. 2000. A first step towards automated detection of buffer overrun vulnerabilities.. In *Network Distributed Systems Security Symposium (NDSS)*, Vol. 20. 1–15.

[67] David R White, Shin Yoo, and Jeremy Singer. 2015. The programming game: evaluating MCTS as an alternative to GP for symbolic regression. In *Proceedings of the Companion Publication of the Annual Conference on Genetic and Evolutionary Computation (GECCO)*. 1521–1522.

[68] Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. 2017. Statsym: vulnerable path discovery through statistics-guided symbolic execution. In *Proceedings of 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 109–120.

[69] Shuangjie Yao and Dongdong She. 2025. Empc: Effective Path Prioritization for Symbolic Execution with Path Cover. In *2025 IEEE Symposium on Security and Privacy (SP)*. 2995–3013. doi:10.1109/SP61157.2025.00190

[70] Chao-Chun Yeh, Han-Lin Lu, Jia-Jun Yeh, and Shih-Kun Huang. 2017. Path exploration based on Monte Carlo tree search for symbolic execution. In *Proceedings of International Conference on Technologies and Applications of Artificial Intelligence (TAAI)*. 33–37.

[71] Qiuping Yi, Yifan Yu, and Guowei Yang. 2024. Compatible Branch Coverage Driven Symbolic Execution for Efficient Bug Finding. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1633–1655.

[72] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (FSE)*. 26–36.

[73] Jaehan Yoon and Sooyoung Cha. 2024. FeatMaker: Automated Feature Engineering for Search Strategy of Symbolic Execution. *Proc. ACM Softw. Eng.* 1, FSE, Article 108 (July 2024), 22 pages. doi:10.1145/3660815

[74] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security)*. 745–761.

[75] Z3. 2024. *A theorem prover from Microsoft Research.* https://github.com/z3prover/z3

[76] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 631–644.

[77] Yiru Zhao, Xiaoke Wang, Lei Zhao, Yueqiang Cheng, and Heng Yin. 2022. Alphuzz: Monte carlo search on seed-mutation tree for coverage-guided fuzzing. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*. 534–547.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009