Appendix: Supplementary Material

A Individual Type-Unsafe Pointer Coverage Improvements Achieved by VITAL

To further understand the unsafe pointer covering capability of VITAL, we dig deeply into the individual improvement achieved by VITAL. Figure 6 shows the overall results, where the *x-axis* represents the number of utilities and the *y-axis* indicates the percentage number of improvements. From the figure, we can observe that VITAL overall outperforms all comparative search strategies (we omitted the results of CBC as VITAL outperforms it over all utilities). In a large portion of cases, VITAL produces significant (up to 3500% under `nurs:icnt`) improvements. This is mainly because the unsafe pointer-guided MCTS excels at navigating the best possible exploration-exploitation trade-offs, thus exploring the path where the number of unsafe pointers is maximized. For a small number of cases (especially for the utilities under `bfs`), VITAL fails to cover more unsafe pointers. We investigated more on such cases and found that this follows one of the characteristics of the MCTS (Section 3.4.3 in [6]): MCTS favors more promising nodes and leads to an asymmetric tree over time. For test programs whose execution tree is symmetric, MCTS may miss a few nodes on the execution tree, thus missing certain unsafe pointers. However, such a characteristic is also advantageous in helping Vital to explore deeper paths, increasing the likelihood of vulnerability detection. Our experiments presented in Section 4.2 also showed that Vital is overall significantly better in terms of memory error detection capability (e.g., up to 57.14% more memory errors are detected).

B Detailed Evaluation Results of CBC [65] on CVE Vulnerability Reproduction

Since the experimental results reported by the original paper (Table 2 in CBC [65]) use *fine-tuned* settings (the values for their customized "–reverse-limit" and "–states-limit" options) for every vulnerability in every search strategy, which we think such a running setting might be an unfair comparison. To reduce fairness threats, we used the same various values used in the CBC paper for the two options and conducted thorough experiments on those values to investigate their impacts. We set a timeout of 1 hour and record the time for every vulnerability per search strategy.

Table 4 presents the detailed results. We can observe that there are large variations of the results under different settings. We selected the settings ("–reverse-limit=5" and "–states-limit=80") that can produce the best overall results and reported them in this paper. For the *Timeout* cases, we performed another run with an increased timeout of 24 hours for a fair comparison.

C Detailed Analysis of the New Vulnerability

C.1 Vulnerability Details

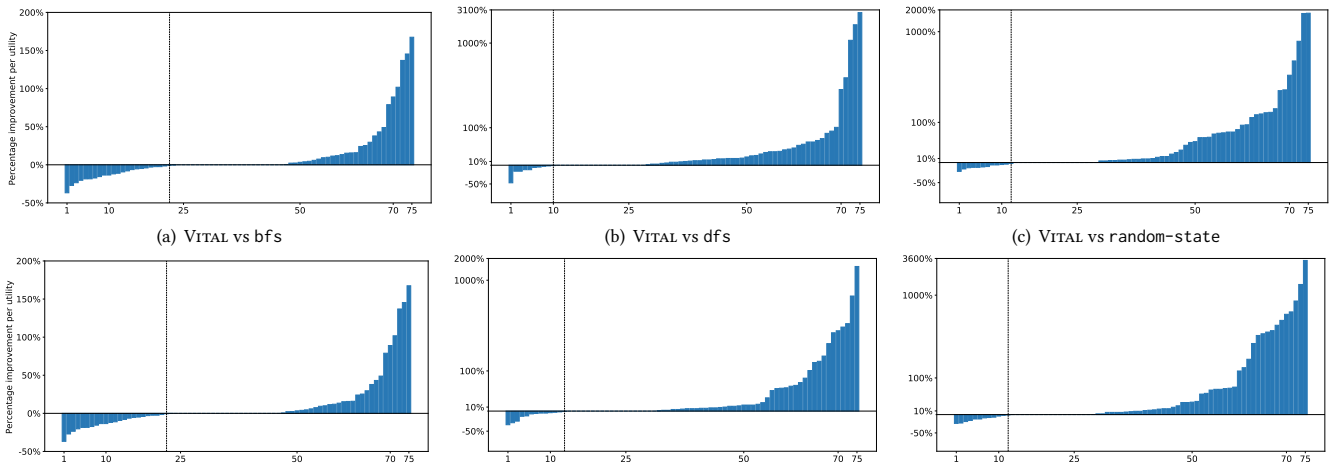
The simplified vulnerable code snippets adopted from `objdump.c` are shown in Listing 1, where a memory object *param->info* (Line 60) is leaked due to improper memory management (i.e., it is allocated but never de-allocated when the execution terminates)[8]. The vulnerable point happens in the function `do_display_target` starting from Line 52, which is initially invoked by the caller function `display_info` (Line 34) in the `main` function. After a deeper investigation of the execution flow of the memory leak, we summarized three requirements that a symbolic execution engine should satisfy to make the execution reach the vulnerable point.

```
1  static bool formats_info;
2  int main (int argc, char **argv) {
3    int c; bool seenflag = false;
4
5    expandargv (&argc, &argv);
6
7    while ((c = getopt_long (argc, argv,
8               "CDE:FGHI:LM:P:RSTU:VW::Zab:defghij:lm:prstvwxz",
9               long_options, (int *) 0)) != EOF) {
10     switch (c) {
11       case 0: break;
12       case 'm': mainchine = optarg; break;
13       case 'M':{
14         char *options; //usafe pointer
15         if (disassembler_options)
16             options = concat (disassembler_options, ...);
17         else
18             options = xstrdup (optarg);
19         free (disassembler_options);
20         disassembler_options =
21                remove_whitespace_and_extra_commas (options);
22         if (!disassembler_options)
23             free (options); //usafe pointer
24       }
25       break; }
26       // ...
27       case 'i':
28         formats_info = true;
29         seenflag = true;
30         break;
31     }
32   }
33   if (formats_info)
34     exit_status = display_info ();
35   else {
36       ...//
37     }
38 } /* adapted from binutils/objdump.c */
39
40 int display_info (void) {
41   struct display_target arg;
42   display_target_list (&arg);
43   ...//
44   return arg.error;
45 } /* adapted from binutils/objcomm.c */
46
47 static void display_target_list (struct display_target *arg) {
48   .../
49   bfd_iterate_over_targets (do_display_target, arg);
50 } /* adapted from binutils/objcomm.c */
51
52 static int do_display_target (const bfd_target *targ, void *data){
53   struct display_target *param = (struct display_target *) data;
54   param->count += 1;
55   size_t amt = param->count * sizeof (*param->info);
56   if (param->alloc < amt){
57     size_t size = ((param->count < 64 ? 64 : param->count)
58         * sizeof (*param->info) * 2);
59
60     param->info = xrealloc (param->info, size); // leaked object
61     ...//
62   }
63   ... //
64 } /* adapted from binutils/objcomm.c */
```

Listing 1: Vulnerable code snippets (memory leak at Line 60)

- #1. The engine should successfully bypass the function `expandargv` (includes input-independent loops) at Line 5.

[8] https://sourceware.org/bugzilla/show_bug.cgi?id=xxx (omit id for anonymity)



(a) VITAL vs `bfs` (b) VITAL vs `dfs` (c) VITAL vs `random-state`

(d) VITAL vs `nurs:covnew` (e) VITAL vs `nurs:md2u` (f) VITAL vs `nurs:icnt`

Figure 6: Individual improvement in terms of covered unsafe pointers achieved by VITAL over 75 utils in `GNU Coreutils`

- #2. The engine should successfully return from the function `getopt_long` (includes input-independent loops and requires multiple execution over a loop) in the while-loop (Line 7).
- #3. The return value from the function `getopt_long` should be the character 'i', so that the value of *formats_info* (defined as global variable at Line 1, assigned in a *switch-case* statement at Line 28, and used in the *if* condition at Line 33) should be *true*.

It would be worthy noting that only if the above requirements are satisfied, the branch of *if*-condition could be *true* at Line 33, and the vulnerable function `display_info` could be invoked to trigger the memory leak issue.

C.2 Why Comparative Approaches Missed it?

Existing path search strategies failed to detect the issue mainly due to the lack of prior expert knowledge or technical support to set up chopped symbolic execution, limited handling of input-dependent loops (not shown due to page limit), and restricted (i.e., not vulnerability-oriented) guided path search heuristic. Due to the above reasons, the comparative tools failed to detect the issue by giving a running timeout of 24 hours.

To set up `Chopper` [57], users need to specify the skipped functions to detect possible vulnerabilities, which could be extremely hard as users have no ideas on how to select skipped functions to detect new vulnerabilities. The only way users might do it is to try different combinations of skipped functions, which tends to be time-consuming and ineffective. Worse still, skipping only at coarse-grain-level code snippets on functions will not help to detect the memory leak in this case as all the functions listed in Listing 1 are responsible for reproducing the issue. Fine-grained levels (e.g., statement or code block) should be supported to skip the execution of certain input-dependent loops. In summary, the above issues of `Chopper` make it difficult to meet the first two requirements.

Baseline approach KLEE [9] is simply restricted by handling input-dependent loops in functions `expandargv` and `getopt_long` (not shown due to page limit) before reaching the vulnerable function `display_info`, so compiling with requirements #1 and #2 within KLEE is hard. It may be easy to bypass these loops with certain bounds, but selecting the optimal bound to expose the issue can be challenging, as only reaching a specific number of iterations will reach the vulnerable point.

CBC [65] bypasses the input-dependent loops successfully by pruning redundant paths that have no new contributions to branch coverage, so that requirements #1 and #2 are easily satisfied. However, as we emphasized before, achieving the best coverage has little to no correlation with the conclusion that there are no bugs. More importantly, some subtle memory issues can only be triggered when the loop is executed with a specific number of iteration times [25]. In this case, to satisfy the requirement #3, the engine should execute a loop at least 25 times. To be specific, the loop inside the function `getopt_longong` scans the long string "*CDE:FGHI:LM:P:RSTU:VW::Zab:defghij:lm:prstvwxz*" one by one and returns the corresponding character iteratively. Only the character 'i' (at position 25 among all characters) is iterated and returned that can make the memory leak happen. Since the loop body was covered in terms of code coverage, CBC gives a very low priority to execute the covered loop again, thus missing the detection of the memory leak issue.

C.3 How Does VITAL Detect it?

VITAL discovered the vulnerable path[9] beneficial from the new indicator for approximating the vulnerable paths and the novel search using a variant of MCTS algorithm. To meet the first two requirements, VITAL skips the *unimportant* input-dependent loops that have no contributions to the accumulation of unsafe pointer coverage by using simulation optimization strategies (details described in Section 3.2.3). To explore the loops that iteratively scans the long string "*CDE:FGHI:LM:P:RSTU:VW::Zab:defghij:lm:prstvwxz*", the simulation in MCTS evaluates the reward of each iteration of the vulnerability-relevant loop, and continues the iteration when the simulation process covers new unsafe pointers (e.g., the ones in Lines 14 and 23). When the iteration is reaching 25 times, the vulnerable function `display_info` is executed and the issue is eventually detected in the function `do_display_target` at Line 60.

[9] VITAL did not directly detect the memory leak, but it is the unique vulnerable path covered by VITAL triggers the memory leak issue.

Table 4: Results of CVE vulnerability reproduction ("–reverse-limit" and "–states-limit" are two customized options that CBC [65] provides for controlling the limit of traversing the dependence graph).

Vulnerablity	Search	reverse-limit	states-limit							
			10	20	30	40	50	80	1000	5000
CVE-2012-1569	Random	4	N/A	N/A	0:44	0:57	0:52	02:03	46:20	49:01
		5	N/A	N/A	N/A	0:58	0:47	01:01	45:13	Timeout
		50	N/A	N/A	N/A	01:41	01:05	02:22	46:58	52:01
	DFS	4	N/A	0:33	0:21	01:44	0:57	02:26	02:26	02:27
		5	N/A	0:32	0:22	0:48	0:57	0:48	01:28	01:27
		50	N/A	0:32	0:24	0:49	0:59	0:56	0:56	0:57
	Coverage	4	N/A	N/A	N/A	0:26	0:36	0:28	0:53	01:07
		5	N/A	N/A	N/A	N/A	0:37	01:18	02:17	01:58
		50	N/A	N/A	N/A	N/A	N/A	01:09	01:19	06:06
CVE-2014-3467(1)	Random	4	N/A	N/A	N/A	N/A	N/A	N/A	Timeout	Timeout
		5	N/A	N/A	N/A	N/A	N/A	N/A	Timeout	Timeout
		50	N/A	N/A	N/A	N/A	N/A	N/A	Timeout	Timeout
	DFS	4	N/A	N/A	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
		5	N/A	N/A	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
		50	N/A	N/A	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
	Coverage	4	N/A	N/A	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
		5	N/A	N/A	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
		50	N/A	N/A	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
CVE-2014-3467(2)	Random	4	N/A	N/A	0:28	0:23	N/A	0:40	04:45	Timeout
		5	N/A	N/A	0:20	0:18	N/A	0:29	04:38	Timeout
		50	N/A	N/A	0:32	0:29	N/A	0:43	05:56	Timeout
	DFS	4	0:05	02:09	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
		5	0:05	02:01	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
		50	0:06	01:56	Timeout	Timeout	Timeout	Timeout	Timeout	Timeout
	Coverage	4	N/A	N/A	0:10	0:35	0:22	0:30	01:08	01:53
		5	N/A	N/A	0:15	0:17	0:19	0:23	01:29	06:13
		50	N/A	N/A	0:24	N/A	0:25	0:33	01:25	0:53
CVE-2014-3467(3)	Random	4	N/A	N/A	N/A	N/A	0:34	0:41	Timeout	Timeout
		5	N/A	N/A	0:29	N/A	0:34	0:54	Timeout	Timeout
		50	N/A	N/A	N/A	0:56	0:45	0:57	Timeout	Timeout
	DFS	4	N/A	0:06	Timeout	Timeout	0:09	0:17	0:19	0:19
		5	N/A	0:07	Timeout	Timeout	0:10	0:18	0:18	0:19
		50	N/A	0:10	Timeout	Timeout	0:12	0:20	0:21	0:22
	Coverage	4	N/A	N/A	0:35	0:41	01:53	Timeout	16:24	02:25
		5	N/A	0:13	0:43	0:30	N/A	Timeout	Timeout	Timeout
		50	N/A	0:13	0:40	N/A	N/A	0:51	Timeout	50:11
CVE-2015-2806	Random	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A	15:21
		5	N/A	N/A	N/A	N/A	N/A	N/A	N/A	14:40
		50	N/A	N/A	N/A	N/A	N/A	N/A	N/A	16:24
	DFS	4	N/A	N/A	N/A	N/A	N/A	20:08	Timeout	Timeout
		5	N/A	N/A	N/A	N/A	N/A	21:48	Timeout	Timeout
		50	N/A	N/A	N/A	N/A	N/A	21:55	Timeout	Timeout
	Coverage	4	N/A	N/A	N/A	N/A	N/A	N/A	02:08	Timeout
		5	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Timeout
		50	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Timeout
CVE-2015-3622	Random	4	N/A	0:43	N/A	0:45	02:03	02:48	05:37	04:13
		5	N/A	0:41	N/A	0:47	01:53	02:40	06:09	04:39
		50	N/A	0:55	N/A	01:03	02:31	03:27	07:29	05:28
	DFS	4	N/A	N/A	06:13	01:30	01:25	01:40	01:35	01:36
		5	N/A	N/A	06:15	01:32	01:26	01:40	01:35	01:37
		50	N/A	N/A	06:53	01:53	01:48	02:07	02:07	02:07
	Coverage	4	N/A	N/A	N/A	N/A	N/A	01:06	14:25	09:58
		5	01:14	N/A	N/A	N/A	02:12	01:19	16:14	13:21
		50	N/A	0:37	01:23	01:04	01:04	01:19	15:49	14:16

* The time format is *minute:second* and "N/A" refers to the normal termination without reproducing the vulnerability. The index number n in CVE-2014-3467(n) represents a distinct location of the vulnerability manifested.