2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 979-8-3315-8582-2/26/$31.00 ©2026 IEEE | DOI: 10.1109/SANER67736.2026.00114

Static Analysis Traces *can* help Dynamic Symbolic Execution: a Replication Study

Sriteja Kummita*, Fabian Schiebel*, Eric Bodden*, Miao Miao†, Shiyi Wei†
* *Heinz Nixdorf Institute, Paderborn University, Fraunhofer IEM, Paderborn, Germany*
{sriteja.kummita, fabian.schiebel, eric.bodden}@uni-paderborn.de
† *University of Texas at Dallas, Richardson, TX, USA*
{mmiao, swei}@utdallas.edu

***Abstract*—Due to necessary approximations and lacking runtime information, static analysis produces false positives. To filter out these false positives, Busse et al. developed a methodology to validate static analysis warnings using dynamic symbolic execution: the authors used the trace information from Clang Static Analyzer and Infer to guide the dynamic symbolic execution engine, KLEE. However, the authors concluded that the static analysis tools could not identify complex use-after-free or null-pointer dereference vulnerabilities, and the trace information for the other simpler vulnerabilities was unhelpful in accelerating KLEE: the runtime of KLEE with just knowing the potential vulnerability location was as fast as using the full static-analysis trace.**

This is counter-intuitive, which is why this paper replicates the study conducted by Busse et al. to validate or refute these findings. It extends the original experiments with trace information extracted from additional static analysis tools based on the PhASAR framework. The replication further adds to the Coreutils benchmark provided in the artifact from Busse et al. a second benchmark: FeatureBench, a synthetic benchmark with configurable control-flow and data-flow complexities designed to evaluate greybox fuzzers. In contrast to Clang Static Analyzer and Infer, the PhASAR-based tools are able to identify *all* the injected vulnerabilities in both the benchmarks, greatly enhancing the number of evaluated static-analysis traces.

This replication yields results that support a conclusion different from the original one obtained by Busse et al.: the traces from the PhASAR-based tools and Clang Static Analyzer actually *do* help KLEE in detecting the vulnerabilities approximately $1.4\times$ faster when compared to KLEE using just the vulnerability location. Interestingly, though, we also observed that when using static-analysis traces KLEE confirms fewer vulnerabilities in Coreutils—the trace information restrains KLEE. Further research is needed to explain and hopefully rectify this effect.

***Index Terms*—Static Analysis, Dynamic Symbolic Execution, Path Sensitivity**

I. Introduction

Static analysis (SA) computes properties of a program without executing it and reasons about program behaviors, such as presence or absence of security vulnerabilities. Extensible SA frameworks include SootUp [1] for Java, and PhASAR [2] for `C` and `C++`. SA tools include Clang Static Analyzer (CSA) [1], Infer [3], Flowcheck [4], etc. Most of these tools also provide trace information to help SA users to better understand the reported security vulnerabilities. The trace information usually contains the potential sequence of runtime operations required to be executed to reach the reported vulnerability location [5]. Such traces are useful in different application contexts such as program understanding, debugging, software testing, and program parallelization [6], [7], [8], [9]. However, due to the lack of runtime information and necessary approximations during the analysis, SA tools produce false positives.

To filter out false positives among the SA reports, Busse et al. [10] investigated the combination of SA with dynamic symbolic execution (DSE). The authors used the trace information extracted from CSA and Infer to guide KLEE [11], a DSE engine for `C` and `C++` that is based on LLVM [12]. Their investigation resulted in two negative results: (1) in real-world applications CSA and Infer could not find any vulnerabilities, yet reported many false positives. In a synthetic benchmark, where ten applications from the GNU Core Utilities (Coreutils) [13] were injected with use-after-free (UAF) and null-pointer dereference (ND) vulnerabilities, the tools detected fewer of these injected vulnerabilities; (2) the trace information extracted from CSA and Infer on the subset of injected vulnerabilities in the Coreutils benchmark was not helpful in accelerating KLEE: only using the vulnerability location as guidance proved as effective as using the full trace.

Such a result is surprising. One would think that the traces should assist DSE in confirming vulnerabilities more quickly. Possibly, the negative result may just be due to the bad quality of CSA's and Infer's SA traces, or maybe the experiments were not comprehensive enough. This paper thus replicates the study conducted by Busse et al. to more deeply investigate the idea of using SA traces in guiding DSE. The study we present extends the original experiments in two ways. First, we implemented two new SA tools, `PSR-UAF` and `PSR-ND`, and extended them to provide trace information for each reported UAF and ND warning. `PSR-UAF` and `PSR-ND` are based on PhASAR, a well-known static analysis framework. Second, our evaluation comprises not only the Coreutils benchmark provided in the artifact from Busse et al. [14] but also a new benchmark based on FeatureBench [15]. FeatureBench is a synthetic benchmark that helps evaluating greybox fuzzers [16] based on program features that describe the control-flow and data-flow complexity of the programs. It is useful for our evaluation as the control-flow complexity influences the path exploration capability of DSE, and the data-flow complexity

[1] https://clang-analyzer.llvm.org/

2640-7574/26/$31.00 ©2026 IEEE
DOI 10.1109/SANER67736.2026.00114

influences the constraint resolution capability of DSE.

The extended evaluation shows that PhASAR-based tools perform better than CSA in that it identifies *all* the injected vulnerabilities in *both* benchmarks.[2] Compared to the original study, this greatly enhances the number of static-analysis traces that one can use for evaluation. When combined with KLEE, we observed that the trace information from PhASAR actually *can help* the DSE exploration to reach these vulnerabilities faster; even trace information from CSA proves to aid KLEE's exploration at least for a subset of the injected vulnerabilities in Coreutils, contradicting the results obtained by Busse et al. on their original benchmark. However, surprisingly, in some cases in Coreutils the use of trace information seems to prevent KLEE from confirming the given vulnerability in the given time budget. We estimate that the trace information may confuse DSE in those cases, causing too much time to be spent on paths that are unhelpful in confirming the vulnerability. Further research with even more and deeper experiments will be required to clearly understand—and hopefully rectify—this effect.

To summarize, this paper's contributions include:

1) a replication study that investigates the usefulness of SA traces in guiding the DSE engine, KLEE,
2) an implementation of `PSR-UAF` and `PSR-ND`, new SA tools that can identify UAF and ND vulnerabilities and produce traces that can be combined with KLEE,
3) an extensive evaluation of the approach using two benchmarks injected with UAF and ND vulnerabilities, and
4) a full artifact comprising our implementation and benchmarks for further reproduction and replication [17].

II. Background

In this section, we first summarize the original study from Busse et al. [10], and then detail the necessary background information relevant to our approach.

1) Original Study from Busse et al.: Busse et al., investigated whether the traces produced by the two SA tools: CSA and Infer, can be used to aid the DSE engine, KLEE and verify the SA warnings. For the rest of this paper, we refer to the manuscript by Busse et al. as *base study*.

Their approach involves three steps: 1) extract static traces for each reported SA warning; 2) communicate the trace information to KLEE by instrumenting `klee_assume` function calls in the program; 3) execute the instrumented program with KLEE to verify the corresponding SA warning.

Figure 1a shows an example use-after-free (UAF) vulnerability. The vulnerability can be detected when all the four events (in lines 4, 7, 10, and 15) are executed at runtime. The corresponding example SA trace is shown in Table I. Each line in the trace contains three elements: line number, step number and a message that specifies which branch (`True` or `False`) to follow at runtime to verify the vulnerability.

[2]Busse et al. already showed that DSE performed much worse with traces by Infer than with traces by CSA. We thus did not replicate any experiments with Infer.

TABLE I: Example SA Trace for the program in Figure 1a.

Line number	Step number	Message
2	1	Taking true branch
5	2	Taking true branch
8	3	Taking true branch
11	4	Taking false branch

At each line specified in the SA trace, a corresponding call to `klee_assume` function in instrumented (lines 2, 5, 8, 11 in Figure 1b and the corresponding implementation in Figure 1c). `klee_assume` function takes two parameters: step number and the branch evaluation condition, which are communicated to KLEE during DSE exploration. Note that, based on the message, a negation is added before the branch evaluation condition (line 17 in Figure 1c).

The step number is used in the *Targeted* search heuristic that guides KLEE along the SA trace using call-graph and control-flow graph distances. It configures KLEE to terminate all paths that do not reach SA trace locations and prioritizes the exploration of the SA trace step sequence. In contrast, the *Default* search heuristic guides KLEE based on the increase in runtime coverage (Section 3.3 in the base study [10]).

The `klee_assume` functions also impose restrictions on the underlying constraint solver using the second parameter, i.e., they ensure that the DSE exploration is only continued if a satisfiable constraint is found. However, if the SA trace is incorrect, the constraint solver is overly-restricted which leads to vulnerability not being found. Hence, three strategies are introduced in the base study to specify the constraint solver resolution restrictions in KLEE (Section 3.2 in the base study [10]): *Ignore* tells KLEE to ignore all the function calls to `klee_assume`, i.e., ignore all the constraints specified in the SA trace; *Require* instructs KLEE to terminate any path in which the path constraint is not satisfiable, i.e., strictly consider all the constraints specified in the SA trace and terminate the exploration when the constraints deem unsatisfiable; *Try* is less restrictive when compared to *Require* as it configures KLEE to only use the satisfiable path constraints in the SA trace, i.e., the DSE exploration is still continued to avoid over-constraining the constraint solver during DSE exploration.

Multi-event vulnerabilities. A UAF vulnerability involves the following four events: (1) heap-allocating an object, (2) creating aliases of the pointer that holds the object's address, (3) freeing the dynamically allocated memory, (4) de-referencing one of the aliased pointers that points to the freed memory.

Busse et al. suggested the following terminology, which distinguished four types of multi-event UAF vulnerabilities depending on the number of aliases of the heap-allocated object[3] (event 2): 1-event vulnerability contains all the events except alias creation in a single source code block. 2-event vulnerability contains heap-object allocation in a different source code block before the last two events. There is no

[3]Specifically, the implementation of Busse et al [14] distinguishes the multi-event vulnerability type based on the number of aliases of the heap-allocated object.

```
1  void UAF_Example(char *data, long
       size) {
2   if(u64(data) == sum(data+8,8)){
3    // event 1
4    ztmp = (int *)
         malloc(sizeof(int));
5    if(u32(data+16) ==
         average(data+20,16)){
6     // event 2
7     ytmp = ztmp;
8     if(u16(data+36) ==
           product(data+38,4)){
9      // event 3
10     xtmp = ytmp;
11     if(sum(data+42,2) ==
           sum(data+44,2)){
12      ...
13     } else {
14      // event 4
15      free(ztmp); printf("%d",
            *xtmp);
16     }
17    } else { ... }
18   } else { ... }
19  } else { ... }
20 }
```

(a) Example program with UAF vulnerability.

```
1  void UAF_Example(char *data, long
       size){
2   if(INSTR_LINE_2(u64(data) ==
         sum(data+8,8))){
3    // event 1
4    ztmp = (int *) malloc(sizeof(int));
5    if(INSTR_LINE_5(u32(data+16) ==
         average(data+20,16))){
6     // event 2
7     ytmp = ztmp;
8     if(INSTR_LINE_8(u16(data+36) ==
           product(data+38,4))){
9      // event 3
10     xtmp = ytmp;
11     if(INSTR_LINE_11(sum(data+42,2)
           == sum(data+44,2))){
12      ...
13     } else {
14      // event 4
15      free(ztmp); printf("%d", *xtmp);
16     }
17    } else { ... }
18   } else { ... }
19  } else { ... }
20 }
```

(b) Instrumented example program based on the example SA trace in Table I.

```
1  int INSTR_LINE_2(bool cond){
2      klee_assume(1, cond);
3      return cond;
4  }
5
6  int INSTR_LINE_5(bool cond){
7      klee_assume(2, cond);
8      return cond;
9  }
10
11 int INSTR_LINE_8(bool cond){
12     klee_assume(3, cond);
13     return cond;
14 }
15
16 int INSTR_LINE_11(bool cond){
17     klee_assume(4, !cond);
18     return cond;
19 }
```

(c) `klee_assume` statements for the instrumented example program in Figure 1b based on the SA trace information in Table I.

Fig. 1: Example program detailing the source code instrumentation that communicates the SA trace information to KLEE.

aliasing, the variable being freed is the one that was also malloced at event 1. 3-event vulnerability contains heap-object allocation and the creation of a single pointer alias in different source code blocks before the last two events, and 4-event vulnerability contains heap-object allocation and two pointer aliases in different source code blocks before the last two events (Figure 1a shows an example).

Similarly, a null-pointer dereference (ND) vulnerability involves the following three events: (1) assigning NULL to an object, (2) creating aliases of the NULL object, (3) dereferencing one of the aliased pointers that is NULL-assigned. The four types of multi-event ND vulnerabilities are similar to the multi-event UAF vulnerabilities except the free call is replaced with an alias to the NULL-assigned object.

Base study benchmarks. Busse et al., first evaluated the approach to verify SA warnings reported by CSA and Infer on 25 real-world applications and CoREBench [18]. The authors then manually analyzed a subset of SA warnings generated by these two SA tools and found that most of them were false positives (Section 4 in the base study [10]). As real-world evaluation was not possible, the authors resorted to a synthetic benchmark by injecting multi-event UAF and ND vulnerabilities into Coreutils. In total, the benchmark contains 297 1-event, 632 2-event, 478 3-event, and 357 4-event UAF and ND vulnerabilities injected across ten Coreutils applications.

2) Taint Analysis: A taint analysis is a versatile data-flow analysis that tracks so-called *tainted* data originating from specified *sources*. It reports a *taint-leak*, if such tainted data may flow into a specified *sink*, without being *sanitized* with a proper sanitizer before. Taint analyses can be used to detect various kinds of bugs including security vulnerabilities, such as use-after-free (UAF), null-pointer dereference (ND), etc. Such kind of errors can be detected both statically or dynamically; this paper focuses on static taint analysis.

To achieve a precise inter-procedural static taint analysis, the underlying algorithm needs to be flow-, context-, and field-sensitive; otherwise, it cannot follow the statement order, match function-returns to the correct call-sites, or distinguish different fields of the same object. Commonly, static taint analyses are solved using the inter-procedural finite distributive subsets (IFDS) [19] algorithm, which reduces the data-flow to a graph-reachability problem and supports flow- and context-sensitive analysis out-of-the-box.

PhASAR already provides an IFDS-based taint analysis. It achieves field-sensitivity by extending the IFDS taint tracking with k-limited field-access paths [20]. As PhASAR operates on LLVM IR [12], it can analyze arbitrary C and C++ applications, among others. Our tool, `PSR-UAF` configures PhASAR's taint analysis to detect UAF vulnerabilities, by specifying the LibC function `free` as source and any memory load operation as sink; there are no sanitizers. Our tool `PSR-ND` configures PhASAR's taint analysis to detect ND vulnerabilities, by specifying NULL-assignment statements as source and any memory load operation as sink. Section III-B further details on our extensions to PhASAR for generating traces.

Clang Static Analyzer performs a path-sensitive analysis on the source-code level. Similar to PhASAR, it builds an exploded graph containing all explored intermediate program states; in contrast to PhASAR, it does not unconditionally merge program states at control-flow merge points (e.g., after an if-else block). Instead, it (1) tries to evaluate branch conditions to prune the state space, (2) unrolls loops at most twice, and (3) relies on program-state-caching to avoid redundant computation. In its

default configuration it tries to follow function calls within the same translation unit until reaching one of multiple thresholds (e.g., stack depth); in contrast, PhASAR always considers the whole program part reachable from the entry-points. When detecting a vulnerability, CSA tracks the most recent explored exploded super graph node and follows the graph edges back to generate a trace; doing so, it reports the shortest trace to the vulnerability.

III. Approach

Befitting a replication study, we follow the approach from Busse et al. in the original work (Section II-1), but on extended benchmarks and with additional SA tool traces. Here, we first detail the extended benchmarks and our SA trace generation before providing a comparison with the base study.

A. Benchmark

There is no *real-world* benchmark that aids the evaluation of DSE tools. Hence, similar to the base study, we resorted to synthetically injecting multi-event UAF and ND vulnerabilities into two benchmarks: Coreutils [14] (from the base study) and FeatureBench [15].

Multi-event UAF and ND vulnerabilities. In contrast to the base study, we use the following three events to represent a UAF vulnerability: (1) heap-allocating an object, (2) freeing the dynamically allocated memory, (3) de-referencing the object that points to the freed memory.

Such a representation is consistent with the multi-event ND vulnerabilities from the base study, where aliases are only used when there are insufficient vulnerability events: 3-event and 4-event ND vulnerabilities. In our benchmark, we use aliases only for 4-event UAF vulnerability whereas the base study uses them for both 3-event and 4-event UAF vulnerabilities.

Therefore, in our benchmark, we distinguish four types of multi-event UAF vulnerabilities as follows: 1-event vulnerability contains all the three events in a single source code block. 2-event vulnerability contains heap-object allocation along with freeing the allocated memory and de-referencing the freed memory in different source code blocks. 3-event vulnerability contains all the three events in different source code blocks. 4-event vulnerability involves a single pointer alias to the heap-allocated object and hence four events in different source code locations: the heap-object allocation, pointer alias to the allocated object, freeing the pointer alias, and de-referencing the object pointing to the freed memory. The multi-event ND vulnerabilities are similar to the base study.

1) Coreutils: The original artifact from the base study contains 297 1-event, 632 2-event, 478 3-event, and 357 4-event UAF and ND vulnerabilities injected across 10 Coreutils applications. However, we observed that the two applications *join* and *csplit* contained more injected vulnerabilities when compared to the other applications: together they account to 75% of the total injections. To reduce the biasing of the results, we reduced the number of vulnerabilities in *join* and *csplit*: the new number is derived by taking the mean of the corresponding number of X-event vulnerabilities in other applications except *join* and *csplit* (e.g. the new number of 1-event vulnerabilities for *join* and *csplit* is derived by taking the mean of all 1-event vulnerabilities except *join* and *csplit* and so on). We then randomly retained the derived number of vulnerabilities for *join* and *csplit*.

TABLE II: Multi-event UAF and ND vulnerabilities in Coreutils benchmark.

Application	1-event		2-event		3-event		4-event		Total	
	UAF	ND	UAF	ND	UAF	ND	UAF	ND	UAF	ND
comm	1	1	6	6	7	7	4	4	18	18
csplit	12	12	11	12	6	10	9	10	38	44
cut	2	2	-	-	-	-	-	-	2	2
env	7	8	6	7	5	5	6	4	24	26
join	10	6	7	5	3	4	1	3	21	18
ln	12	10	15	17	13	13	9	10	49	50
nl	7	9	12	12	9	9	10	9	38	39
od	20	17	5	6	-	-	1	1	26	24
split	16	15	4	5	-	-	-	-	20	20
uniq	7	15	19	5	12	-	9	-	47	20
Total	**94**	**86**	**85**	**89**	**55**	**64**	**49**	**53**	**283**	**292**

Our plan was also to reuse the vulnerability locations (or line numbers) for multi-event vulnerabilities provided in the base study artifact. However, using these line numbers to instrument the vulnerability events resulted in compilation errors as the line numbers were inaccurate. Hence, we manually updated the line numbers either by incrementing or decrementing them until an offset of 5, while ensuring the new line number is in the same lexical scope as the previous line number.

Using these new vulnerability locations, we executed the default KLEE configuration with three different seeds (to counter KLEE's randomness) and only retained the vulnerabilities that are detected by KLEE within a 10 minute timeout and in all the three runs. Table II details the various UAF and ND vulnerabilities across the ten Coreutils applications.[4]

2) FeatureBenchDSE: FeatureBench is a synthetic benchmark to evaluate greybox fuzzing (GF) based on the program features that describe the control-flow and data-flow complexity of the programs [15]. In contrast to Coreutils, evaluation on FeatureBench can also provide deeper insights on the weaknesses in the SA tools with respect to vulnerability detection. Since FeatureBench can only generate programs with 1-event vulnerabilities, we extended its implementation to also generate programs for 2-event, 3-event, and 4-event UAF and ND vulnerabilities for each program feature. In the remainder of the paper we refer to this extended benchmark as FeatureBenchDSE.

FeatureBenchDSE provides configurable parameters to manipulate each program feature. The parameters `Width`, `Depth`, `BBranch`, and `Iteration` define the control-flow features, whereas the parameters `Length` and `Depth` represent the data-flow features.

The `Width` specifies the number of branching paths from each branch condition and the `Depth` controls the nesting level of each condition statements. The `BBranch` parameter specifies on which branch the vulnerability is located. Figure 2

[4]Similar to the base study, we also used *GNU Coreutils 8.31*.

TABLE III: Overview of features in FeatureBenchDSE.

Category	Feature	Description	Parameter	Parameter Settings	Other Settings	# of Programs	Total
Control-Flow	COMW	Number of conditional branches	Width	{4,6,8...14}	Depth=4	6	28
	COMD		Depth	{4,6,8...14}	Width=2	6	
	COMB	Location of the bug branch	BBranch	{16,32,64...256}	Depth=8, Width=2	8	
	RECURD	Recursions with data constraints	Iteration	{50,75,100,...,225}		8	
Data-Flow	MAGICL	Magic bytes	Length	{1,2,3,...8}	Depth=8, Width=1	8	24
	MAGICD	Nested magic bytes	Depth	{4,5,6,...11}	Width=1	8	
	CHECKSUMD	Nested checksum tests		{4,5,6,...11}	Width=1	8	
Total						**52 x 4 types of multi-event bugs = 208**	

shows an example program with these three control-flow parameters.

```
void W$width_D$depth_B$bbranch($param) {
  if ($path_condition_1) { // level 1
    if ($path_condition_2) { … } // level $depth
    else if ($path_condition_3) { … }
  } else if ($path_condition_4) {
    xtmp = (int *)malloc(sizeof(int));
    free(xtmp);
    if ($path_condition_5) { … }
    else if ($path_condition_6) {
      // level $(width^depth)
      printf("%d", *xtmp);
    }
  }
}
```

Fig. 2: FeatureBenchDSE example with parameters width, depth, bug branch.

The parameter `Iteration` controls the recursion depth in the program. Figure 3 provides a corresponding example program. The ND vulnerability is triggered when `i` is equal to `$iteration`. The control-flow parameters influence the path exploration capability of DSE by creating an exponentially increasing number of paths (based on the parameter values), while only one valid path will reach the vulnerability.

```
void Recur_I$iteration(char *data, long
size, int i) {
  if ($path_condition_1) {
    if (i == $iteration_1) {
      // $iteration_1 < $iteration
      xtmp = NULL;
    }
  }
  ...
  if ($path_condition_3) {
    if (i == $iteration) {
      printf("%d", *xtmp);
    }
  }
  Recur_I$iteration(data, size, i+1);
}
```

Fig. 3: FeatureBenchDSE example with parameter iteration.

As for the data-flow parameters, `Length` defines the number of magic characters involved in satisfying the condition, and `Depth` represents the nesting level of condition statements. Figure 4 shows an example program for the feature, (nested) magic bytes.

```
void L$length_D$depth(char *data, long size) {
  if (data[0] == '<' && ... && data[$length-1] == '>') {
    // level 1
    xtmp = (int *)malloc(sizeof(int));
    free(xtmp);
    ...
    if (data[$length*($depth-1)] == '/' &&
        ... && data[$length*$depth-1] == '+') {
      // level $depth
      printf("%d", *xtmp);
    }
  }
  ...
 }
}
```

Fig. 4: FeatureBenchDSE example for (nested) magic bytes.

```
void D$depth(char *data, long size) {
  if ($CHECKSUM_TEST1) {
    // level 1
    xtmp = (int *)malloc(sizeof(int));
    free(xtmp);
    ...
    if ($CHECKSUM_TEST$depth) {
      // level $depth
      printf("%d", *xtmp);
    }
  }
  ...
 }
}
```

Fig. 5: FeatureBenchDSE example for nested checksum tests.

Figure 5 provides an example program of the feature, nested checksum tests. The `Depth` parameter controls the nesting level of `CHECKSUM_TEST`. Each `CHECKSUM_TEST` is a manually crafted condition that defines a data constraint: for example, the sum of 42nd and 43rd byte in the input should be equal to the sum of the next 2 bytes is specified as `sum(data+42, 2) == sum(data+44, 2)`. By creating hard-to-solve branch conditions, the data-flow parameters influence the constraint resolution capability of DSE.

Table III provides the summary of seven features using six parameter configurations in FeatureBenchDSE. In contrast to the original FeatureBench, we only consider a subset of the program features with modified parameter configurations: we ignored the program features from FeatureBench that are more specific to greybox fuzzing. The minimum `Depth` of FeatureBenchDSE programs is set to four to accommodate multi-event vulnerabilities (the columns *Parameter Settings* and

Other Settings in Table III). In summary, FeatureBenchDSE contains 208 synthetic programs for each vulnerability kind (UAF and ND): 52 programs across seven features × 4 multi-event vulnerabilities. For the three control-flow features COMB, COMD, and COMW, the last event is always inserted into the configured branch location and the remaining events are inserted into randomly selected predecessor branches that lead to the configured branch location. For recursive programs, events are injected at different recursion depths compared to the last event which is always injected into the final iteration. For data-flow features, events are injected into branches at different depth levels compared to the last event, which is always injected in the deepest branch.

B. Generating Static-analysis Traces

To generate SA traces, we use the PhASAR-based (Section II-2) taint-analysis tools (`PSR-UAF`, `PSR-ND`) and CSA.[5] Here we explain the trace extraction methodology in PhASAR.

1) Computing SA Traces in PhASAR: To extract concrete vulnerability traces from PhASAR, we augment the IFDS solving process with on-the-fly path tracking inspired from Lerch et al. [21], building an explicit exploded supergraph (ESG) that combines control- and data-flows, while the taint analysis problem is solved. As the IFDS solver needs to follow these flow-edges anyway [19], storing them explicitly in a graph only adds a constant factor of overhead. For each warning reported, we identify the corresponding node in the ESG and follow all (transitively) incoming edges backwards to create an induced sub-graph containing only the control- and data-flows that are relevant for the warning. Importantly, we automatically skip function calls that do not influence the taint flow.

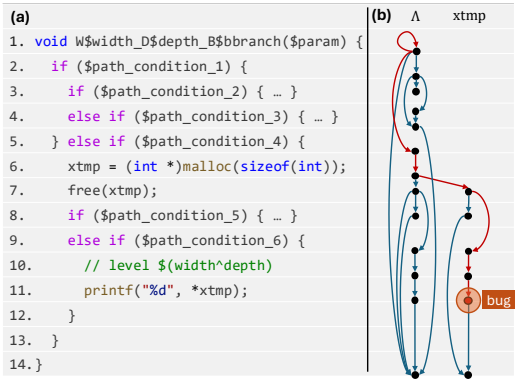


Fig. 6: (a) An example UAF vulnerability. (b) The ESG computed by the taint analysis. The highlighted red edges denote the path to the bug location.

Figure 6a shows the same example program from Figure 2 along with the corresponding ESG generated by `PSR-UAF` in Figure 6b. The UAF vulnerability in line 11 is only exploitable if an attacker manages to set `$path_condition1` and `$path_condition5` to `false`, and `$path_condition4` and `$path_condition6` to `true`. We raise a path-query for the vulnerability location $\langle 11, xtmp \rangle$, following the (transitively) incoming edges in backwards direction to compute the induced sub-graph (highlighted in red in Figure 6b). During the backwards traversal, after reaching the taint analysis source (`free` function at line 7), we follow the incoming edges of Λ, a special tautological fact that only encodes control-flow and no data-flow information.

The sub-graph now contains all (potentially infinitely many) paths, through which the vulnerability location can be reached. Due to loops or recursion, the sub-graph may contain cycles, which prevents naïve unrolling with a simple depth-first graph traversal. To ensure termination of the graph traversal, we limit the number of extracted paths to a constant threshold.[6]

When constructing the ESG in an inter-procedural scenario, the graph looses context-sensitivity by connecting all call-sites with an edge to the called function and connecting all return statements with all return-sites of the returned-from function. To re-construct context-sensitivity and report only valid paths, we maintain a stack of call-sites while performing the depth-first graph-search and only follow the return-edges that match the top call-site in the stack, thereby filtering out invalid paths.

Based on the extracted valid paths, we examine conditional statements that lay on these paths and annotate information about whether the `true` or `false` branch is taken. As our SA tool assumes a constant threshold to unroll loops, we decided to not include the path conditions from inside loops (e.g., test conditions in `for`-loops) into the extracted trace. A runtime execution path may need to visit a loop's body more often than our threshold would allow.

Table IV shows the trace generated by `PSR-UAF` in the same format used in the base study (Table I), for the example shown in Figure 6. As expected, it requires for our example that `$path_condition1` and `$path_condition5` evaluate to `false`, and `$path_condition4` and `$path_condition6` evaluate to `true`.

TABLE IV: Trace from `PSR-UAF` for the example in Figure 6.

Line number	Step number	Message
2	1	Taking false branch
5	2	Taking true branch
8	3	Taking false branch
9	4	Taking true branch

For the Coreutils benchmark (see Section III-A1), we have configured all the SA tools to only analyze the source files of the application and ignore all the library files.[7] Such a configuration ensures that the traces include only the application source code, where all the vulnerability events are present.

2) Scaling SA Trace Extraction in PhASAR: Extracting traces from an ESG via depth-first search can become unscalable, as the number of possible paths grows exponentially

[5]Compared to the base study, we use the updated version 14.0.1 of CSA.

[6]The loop unrolling threshold is defaulted to 1 in our experiments.

[7]Specifically, we only consider the source files in the `src` folder and ignore other files.

TABLE V: Summary of replication study vs. base study.

Category	Base study	Current study
Benchmark	*33 UAF and 22 ND vulnerabilities in Coreutils detected by both:*	*FeatureBenchDSE; 283 UAF vulnerabilities and 292 ND vulnerabilities in Coreutils detected by both:*
SA tools	*CSA, Infer*	*CSA, PhASAR-based tools*
DSE tool	*KLEE 2.1*	*KLEE 3.1*
LLVM	*11.0.1*	*14.0.6*
Z3 SMT Solver	*4.8.8*	*4.8.17*
Constraint solver restriction strategies	Ignore, Require, Try	
Search heuristics	Default, Targeted	
Evaluation configurations	IgnoreDefault, IgnoreTargeted, IgnoreTargetLast, TryDefault, TryTargeted, RequireDefault, RequireTargeted	
Evaluation metrics	time-to-exposure (TTE)	

with the branching depth. Especially, due to merge points in the graph (e.g., merging flows from then- and an else-branch), the path reconstruction may visit the same call-site multiple times, requiring to not only visit all possible sub-paths inside the callee again, but also to repeat the context filtering. To avoid such redundant computations, we introduce a caching mechanism that saves a limited number of valid sub-paths from a call-site to a return-site. This caching mechanism is similar to computing function summaries in the data-flow analysis [19], but as major difference, it stores "summaries" per call-site instead of per callee.

C. Comparison with the Base Study

Table V presents the summary of our replication study, highlighting the differences with the base study in *italics*. The main differences are in the benchmarks (Section III-A), SA tools (Section III-B) and other supplementary tool versions.

Busse et al. already showed that DSE performed much worse with traces by Infer than with traces by CSA. We thus did not replicate any experiments with Infer. We used the latest version of KLEE at the time of writing this paper[8] and updated the implementation corresponding to `klee_assume` instrumentation using the base study artifact. The constraint solver restriction strategies, search heuristics, evaluation configurations and metrics from the base study remain unchanged.

IV. Evaluation

We evaluated our approach using the following research questions:

- **RQ1.** How do the vulnerability detection capabilities of `PSR-UAF` and `PSR-ND` compare to CSA on Coreutils?
- **RQ2.** How does the combination of SA tools and DSE perform on Coreutils?
- **RQ3.** How do the SA tools and their combination with DSE perform on FeatureBenchDSE?

Based on the constraint solver restriction strategies and search heuristics, we have reused the seven evaluation configurations from the base study with *IgnoreDefault* being the default (Table V). *IgnoreTargetLast* is a special case of *IgnoreTargeted* where during the exploration KLEE only uses the last statement in the SA trace, i.e., the vulnerability location. The metric *TTE* records the time taken by the configuration to successfully detect the injected vulnerability. Similar to the base study, TTE does not include the time taken by SA tools to generate a trace as the approach assumes that the traces are available before executing KLEE and aims to confirm the SA warnings.

All our experiments were conducted in Docker containers based on Ubuntu 22.04, running on an Intel(R) Xeon(R) Platinum 8462Y+ processor with 64 GiB RAM. Each container was allocated with 2GiB RAM and 2 cores: one container for each of the seven evaluation configurations. We repeated each experiment three times with different seeds to counter randomness introduced by KLEE. [9] All configurations are executed after extracting the traces from the SA tools: `PSR-UAF`, `PSR-ND` and CSA. To enable a fair comparison between the SA tools, we removed all the line numbers that involve loop conditions from the trace produced by CSA (Section III-B).

A. RQ1. Vulnerability Detection Capability of SA Tools

We use the availability of trace information from the SA tools as a proxy to measure vulnerability detection. Table VI summarizes the vulnerability detection statistics of the SA tools in terms of number of traces produced for the injected UAF and ND vulnerabilities in Coreutils; note that we restrict both tools to report at most one trace per vulnerability.

TABLE VI: Traces generated by the SA tools for the injected vulnerabilities in Coreutils.

	UAF			ND		
	Total	**PSR-UAF**	**CSA**	**Total**	**PSR-ND**	**CSA**
1-event	94	94	76	86	83	70
2-event	85	85	8	89	88	8
3-event	55	55	8	64	64	1
4-event	49	49	2	53	53	1
Total	283	283	94	292	288	80

For both the vulnerability types, PhASAR-based tools (`PSR-UAF` and `PSR-ND`) identify and produce traces for almost *all* the injected multi-event vulnerabilities except for four ND vulnerabilities in Coreutils within the configured time-limit of three minutes. Although, `PSR-ND` detected these four vulnerabilities, it could not generate a trace within the time-limit. This is due to the inherent practical limitation to traverse all the possible paths to reach the vulnerability location in a reasonable amount of time.

CSA produces a smaller number of traces for both UAF and ND vulnerabilities when compared to PhASAR. We can also see that CSA works better for 1-event vulnerabilities and produces slightly more traces for UAF vulnerabilities when compared to ND vulnerabilities. Thus CSA's performance on Coreutils aligns with the one reported in the base study.

[8]https://github.com/klee/klee/releases/tag/v3.1

[9]We used three random seeds that are different from the base study (9005810, 8481115, 87256).

In terms of detecting and producing traces for the injected vulnerabilities in Coreutils, PhASAR-based tools perform significantly better than CSA. The vulnerability detection capability of CSA is consistent with the base study results and makes it particularly unsuitable for identifying vulnerabilities that involve many events.

B. RQ2. Evaluation using Coreutils Benchmark

Our Coreutils benchmark includes 283 UAF vulnerabilities and 292 ND vulnerabilities (Table II). We configured KLEE to use 10 minutes as the timeout and one minute for constraint solver invocations. A vulnerability is marked *success* when the configuration identifies it during DSE exploration within the configured timeout in all the three runs, whereas it is marked *failure* when the configuration either does not detect the vulnerability within the timeout or terminates early without finding the vulnerability in all the three runs.

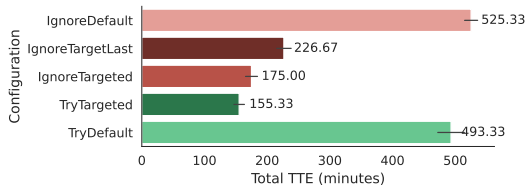
TABLE VII: Total vulnerabilities detected by six configurations for the combination of two SA tools with KLEE in all the three runs.

	UAF (283)		ND (292)	
	PSR-UAF	CSA	PSR-ND	CSA
TryDefault	239	87	260	68
IgnoreTargetLast	220	85	244	73
TryTargeted	212	72	222	63
IgnoreTargeted	200	76	217	65
RequireDefault	108	88	122	71
RequireTargeted	84	49	99	41

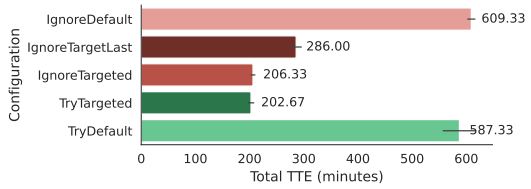
Table VII shows the number of injected vulnerabilities detected across the six evaluation configurations and in all three runs for each SA tool. Note that *IgnoreDefault* is not included here because, the benchmark only contains the vulnerabilities that are successfully detected by this default KLEE configuration (Table II).

Firstly, we can infer that none of the configurations are better than *IgnoreDefault* in detecting the vulnerabilities: all the configurations find less vulnerabilities in comparison with KLEE's default configuration. Second, *TryDefault* detects the highest number of vulnerabilities when KLEE is combined with PhASAR-based SA traces which is followed by *IgnoreTargetLast*. *RequireDefault*, and *RequireTargeted* detects the least number of vulnerabilities. Note that the Require strategy restricts the constraint solver and terminates any path in which the SA trace constraints are not satisfiable. In other words, the traces from PhASAR-based tools do not aid KLEE in constraint solver resolutions. However, *RequireDefault* detects the highest number of UAF vulnerabilities when KLEE is combined with CSA traces which is followed by *TryDefault* and *IgnoreTargetLast* for ND vulnerabilities which is followed by *RequireDefault*.

1) Performance of KLEE combined with PhASAR-based Traces: Since each configuration finds a variable number of vulnerabilities, we compare the total TTE of the common vulnerabilities detected by `PSR-UAF` and `PSR-ND` with KLEE in the five configurations and three runs to judge the performance of each configuration: we omit the results from least performing *RequireDefault* and *RequireTargeted* (see Table VII).



(a) Total TTE of the 186 common vulnerabilities across five configurations and three runs in `PSR-UAF` with KLEE. The horizontal black bars denote the standard deviation.



(b) Total TTE of the 204 common vulnerabilities across five configurations and three runs in `PSR-ND` with KLEE. The horizontal black bars denote the standard deviation.

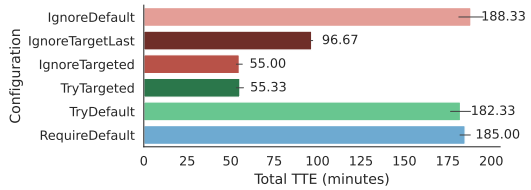
Fig. 7: Performance of KLEE with PhASAR-based traces on Coreutils.

Figure 7a shows the total TTE of the 186 common UAF vulnerabilities and Figure 7b shows the total TTE of the 204 common ND vulnerabilities across the remaining five configurations and three runs. *IgnoreTargetLast*, *IgnoreTargeted*, and *TryTargeted* have varying performance where *TryTargeted* outperformed by detecting the UAF and ND vulnerabilities on average 71 minutes and 83 minutes faster than *IgnoreTargetLast* respectively. However, *TryTargeted* it is only slightly faster than the second best configuration, *IgnoreTargeted* in the case of ND vulnerabilities (Figure 7b).

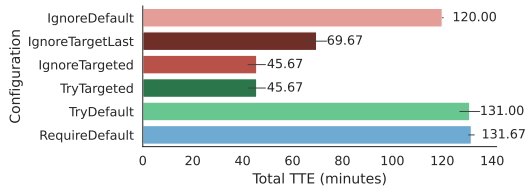
Recall that the Targeted heuristic uses the SA trace sequence during the DSE exploration, i.e., the complete trace from PhASAR-based tools help KLEE to prune the paths and reach the vulnerability faster when compared to just using the last trace statement. We highlight that this inference deviates from the base study, where *IgnoreTargetLast*, *IgnoreTargeted*, and *TryTargeted* had similar performance when evaluated on subset of UAF and ND vulnerabilities with the traces from CSA and Infer. Although, *IgnoreDefault* and *TryDefault* find many vulnerabilities (Table VII), they have similar total TTE and are approximately 2.2× slower in detecting the vulnerabilities when compared to others.

2) Performance of KLEE combined with CSA Traces: We omit the results from the least performing *RequireTargeted* (see Table VII) and only analyze the performance of the common vulnerabilities detected across the remaining six configurations and in three runs.

Figure 8a and Figure 8b show the total TTE of the common UAF and ND vulnerabilities across the six configurations and



(a) Total TTE of 69 common UAF vulnerabilities across six configurations and three runs in CSA with KLEE. The horizontal black bars denote the standard deviation.



(b) Total TTE of 54 common ND vulnerabilities across six configurations and three runs in CSA with KLEE. The horizontal black bars denote the standard deviation.

Fig. 8: Performance of KLEE with CSA traces on Coreutils.

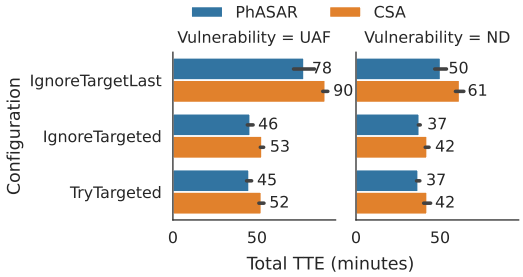


Fig. 9: Total TTE of the common UAF and ND vulnerabilities across *IgnoreTargetLast*, *Targeted configurations between two SA tools. The horizontal black bars denote the standard deviation.

three runs. Similar to PhASAR-based tools, *IgnoreTargetLast*, *IgnoreTargeted*, and *TryTargeted* have varying performance with *IgnoreTargeted* and *TryTargeted* detecting the UAF and ND vulnerabilities on average 41 minutes and 24 minutes faster than *IgnoreTargetLast* respectively.

3) Performance comparison of KLEE combined with SA Traces: To judge the usefulness of SA trace information in pruning paths during KLEE exploration and reaching the vulnerability faster, we compare the total TTE of the common UAF and ND vulnerabilities across the SA tools (PhASAR-based and CSA) between *IgnoreTargetLast*, *IgnoreTargeted* and *TryTargeted*. Note that the *Targeted* heuristic configures KLEE to terminate all paths that do not reach SA trace locations and prioritizes the DSE exploration of the SA trace sequence.

Figure 9 shows the total TTE of 60 UAF and 45 ND vulnerabilities that are detected by all the SA tools and in three runs. We can infer that across all the three configurations, KLEE

TABLE VIII: Feature-wise trace generation statistics of the SA tools on FeatureBenchDSE.

Feature	1-event		2-event		3-event		4-event	
	CSA	PSR*	CSA	PSR*	CSA	PSR*	CSA	PSR*
CHECKSUMD	8/8	8/8	0/8	8/8	0/8	8/8	0/8	8/8
COMB	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8
COMD	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
COMW	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6
MAGICD	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7
MAGICL	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8
RECURD	0/8	8/8	0/8	8/8	0/8	8/8	0/8	8/8

was able to detect the vulnerabilities faster using PhASAR's traces compared to the traces from CSA.

In summary, KLEE's vulnerability detection ability reduces when using SA traces. In contrast to the base study, on the detected vulnerabilities the trace information from both the SA tools help in accelerating KLEE: the *Targeted* search heuristic helps the DSE to find vulnerabilities faster when compared to just using the last trace statement. Further, the traces from PhASAR-based tools help KLEE in detecting the vulnerabilities faster compared to CSA.

C. RQ3. Evaluation using FeatureBenchDSE Benchmark

First, we use FeatureBenchDSE to evalute the vulnerability detection capability of the SA tools across different program features. Table VIII shows the corresponding statistics for seven features in the FeatureBenchDSE benchmark for the two SA tools and multi-event vulnerabilities.[10] It is clear that PhASAR-based tools report all the injected vulnerabilities across all the seven features. However, CSA fails to report all the multi-event vulnerabilities injected in the recursive programs (RECURD) and all the multi-event vulnerabilities except 1-event vulnerabilities in the programs with nested checksum tests (CHECKSUMD): highlighted in pink in Table VIII.

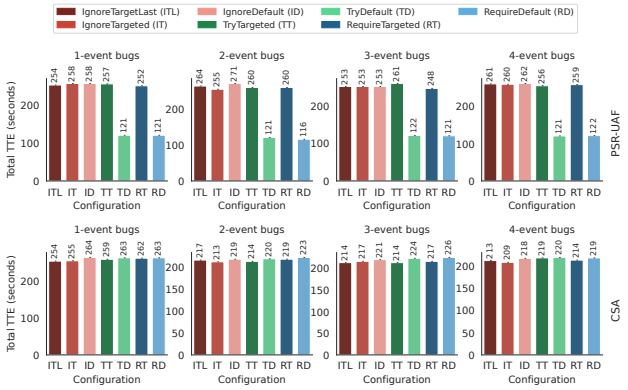


Fig. 10: Total TTE of the multi-event UAF vulnerabilities across sevent configurations between two SA tools. The vertical black bars denote the standard deviation.

[10]The statistics are same for both the injected UAF and ND vulnerabilities and hence we do not differentiate them.

Second, we also evaluate the combination of SA traces with KLEE on the vulnerabilities detected by all the SA tools: hence we omit the features RECURD and CHECKSUMD. We configured KLEE to use 3 minutes as the timeout and one minute for constraint solver invocations. Figure 10 shows the total TTE for all the remaining five features across seven evaluation configurations and two SA tools for all the multi-event UAF vulnerabilities[11]. We can infer that *TryDefault* and *RequireDefault* performs best when KLEE uses PhASAR traces (top four graphs) and there is no substantial benefit when KLEE uses CSA traces (bottom four graphs).

Overall, when KLEE uses PhASAR traces then *TryDefault* and *RequireDefault* detect the vulnerabilities faster than other configurations. CSA does not report most of the injected vulnerabilities on programs with recursion or hard-to-solve branch conditions.

V. Related Work

The recent registered report by Xu et al. [22] is a closely related work to our paper. Coincidentally, it also suggests a reproduction of the experiments by Busse et al. [10]. We first explain the differences with the registered report and then discuss other related works.

A. Comparison with Planned Study by Xu et al.

First, as a registered report, the paper by Xu et al. suggests a study yet to be conducted. As of now, the study is still ongoing and only preliminary results have been published. Further, the registered report is a reproducibility study whereas our work resembles a replication study. A reproducibility study aims at obtaining consistent results using the same experimental methods and data as the original study. A replication study such as ours aims at validating (or refuting) the original results using new experimental data [23].

But Xu et al.'s approach differs from ours in other ways as well. The authors' approach considers improvements in KLEE, whereas our approach is based on new, PhASAR-based, SA tools. The authors suggest two modifications to KLEE: a loop-aware control-flow analysis that instructs KLEE to ignore the constraint solver resolutions for the SA traces involving loop statements, and an update in KLEE's termination criteria to validate if the execution path includes all the SA trace statements. In contrast, our approach does not apply any modifications to KLEE. Instead we configure the SA tools to ignore all the path conditions that are part of the loops.

Also, our evaluation uses benchmarks different from the ones that Xu et al. use. The authors' preliminary evaluation results are based on two benchmarks: the subset of vulnerabilities in Coreutils similar to the base study, and the SV-Benchmark suite [24]. The authors ignored the *IgnoreTargetLast* configuration (Table V) in their current evaluation which contributed to one of the negative results reported in the base study. Whereas, our evaluation is based on two different benchmarks: FeatureBenchDSE and the extended Coreutils benchmark (Section III-A) from the base study; we include all the seven evaluation configurations similar to the base study (Table V).

Considering the above mentioned differences, we argue that both the studies are relevant to investigate the combination of SA and DSE. Once the evaluation suggested in the registered report is published, an interesting future work could be to combine the technical improvements from Xu et al. to evaluate a combination with our SA tools, `PSR-UAF` and `PSR-ND`.

B. Other Relevant Approaches

Recent advances in static analysis (SA) and dynamic symbolic execution (DSE) have focused on reducing false positives in SA reports. *FuzzSlice*, introduced by Murali et al. [25] uses function-level fuzzing to prune false positives in SA reports. The *TOPr* framework, proposed by Niddodi et al. [26] enhances static code pruning to facilitate faster and more precise directed fuzzing. SATRACER by Li et al. [27] combines lightweight symbolic execution with SA to refine the reports, thereby improving the precision of bug detection.

Dynamic symbolic execution has been employed to validate SA reports by pruning infeasible paths. Molina et al. [28] presented a learning-based technique to efficiently discard paths that cannot lead to a reported bug, optimizing symbolic execution's performance. Yi et al. [29] introduced a coverage-driven approach that leverages branch coverage techniques to locate bugs effectively while avoiding redundant paths. On Coreutils programs, their approach atop KLEE resulted in 45% reduction in paths and 3x speedup, thereby attaining higher branch coverage in less time.

VI. Threats to Validity

We have identified the following four threats to validity.

First, our evaluation is based on two synthetic benchmarks and does not include real-world vulnerabilities. We plan to consider such an evaluation in the future, but the sparsity of cases in which such vulnerabilities are documented on the line level complicates the curation of such a dataset.

Second, the program features contained in FeatureBenchDSE might be insufficient to completely judge the influence of each feature on the combination of SA and DSE. To mitigate this threat, we also evaluate our approach on Coreutils.

Third, the trace reported by our PhASAR-based tools might not be the best trace to combine with DSE. Since a statement in the program can be reached from multiple paths, we use the first trace produced by the tools to combine with KLEE. Also, it is practically infeasible to evaluate all the possible traces for a single vulnerability. We also leave this interesting problem of identifying the optimal trace for DSE to future work.

Fourth, the randomness in KLEE can affect the confidence of the results. To mitigate this threat, we conducted the experiments using three different seeds.

VII. Conclusion

In this paper, we report on a study that replicates the one conducted by Busse et al. on combining SA traces

[11]The trend is similar for ND vulnerabilities and hence we do not include it in the paper but is provided in the artifact [17].

with the DSE engine KLEE. Our study adds two SA tools (`PSR-UAF`, `PSR-ND`) and two extended benchmarks: Coreutils and FeatureBenchDSE. Both benchmarks are injected with multi-event UAF and ND vulnerabilities. The results reveal that the PhASAR-based tools have significantly better vulnerability detection capability than CSA: they find all the injected vulnerabilities in our benchmarks. When combined with KLEE, the trace information from both the SA tools aids in accelerating KLEE, especially, in pruning the paths to reach the vulnerability faster (*Targeted* search heuristic). Although, the SA traces help in finding many vulnerabilities faster, we observed that on Coreutils the total number of vulnerabilities confirmed by KLEE using these traces is lower than with the default KLEE without the traces.

Some of our evaluation results deviate from the ones reported by Busse et al.: while CSA and Infer could not detect many injected vulnerabilities in their benchmark, the new PhASAR-based tools identify all of them, yielding a much more comprehensive benchmark. This then shows that using SA traces *can aid* DSE in reaching the vulnerabilities faster than just using the last trace statement.

Acknowledgments

This work was supported in part by the National Science Foundation grant CCF-2047682.

References

[1] K. Karakaya, S. Schott, J. Klauke, E. Bodden, M. Schmidt, L. Luo, and D. He, "Sootup: A redesign of the soot static analysis framework," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 229–247.

[2] P. D. Schubert, B. Hermann, and E. Bodden, "PhASAR: An Inter-procedural Static Analysis Framework for C/C++," in *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer-Verlag, 2019, pp. 393–410.

[3] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–465.

[4] I. Budde, "Exploring the use of static data flow analysis for automatic vulnerability audits of rust code," masterthesis, Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, 2024. [Online]. Available: https://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-83219

[5] T. Eisenbarth, R. Koschke, and G. Vogel, "Static object trace extraction for programs with pointers," *Journal of Systems and Software*, vol. 77, no. 3, pp. 263–284, 2005, software reverse engineering. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121204002079

[6] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, p. 1–36, Mar. 2005. [Online]. Available: https://doi.org/10.1145/1050849.1050865

[7] A. De Lucia, "Program slicing: methods and applications," in *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, 2001, pp. 142–149.

[8] M. Harman and R. Hierons, "An overview of program slicing," *software focus*, vol. 2, no. 3, pp. 85–92, 2001.

[9] M. Chalupa and J. Strejček, "Evaluation of program slicing in software verification," in *Integrated Formal Methods*, W. Ahrendt and S. L. Tapia Tarifa, Eds. Cham: Springer International Publishing, 2019, pp. 101–119.

[10] F. Busse, P. Gharat, C. Cadar, and A. F. Donaldson, "Combining static analysis error traces with dynamic symbolic execution (experience paper)," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 568–579. [Online]. Available: https://doi.org/10.1145/3533767.3534384

[11] C. Cadar, D. Dunbar, and D. Engler, "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*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[12] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. Int. Symp. Code Generation and Optimization (CGO)*. IEEE, 2004, pp. 75–88.

[13] (2022) Gnu coreutils. [Online]. Available: https://www.gnu.org/software/coreutils/

[14] F. Busse, P. Gharat, C. Cadar, and A. F. Donaldson, "Artefact for the issta 2022 paper: Combining static analysis error traces with dynamic symbolic execution (experience paper)," May 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6539575

[15] M. Miao, S. Kummita, , E. Bodden, and S. Wei, "Program feature-based benchmarking for fuzz testing," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '25)*, ser. ISSTA '25. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3728899

[16] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021.

[17] S. Kummita, F. Schiebel, E. Bodden, S. Wei, and M. Miao, "Artifact for static analysis traces can help dynamic symbolic execution," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.18275916

[18] M. Böhme and A. Roychoudhury, "Corebench: studying complexity of regression errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 105–115. [Online]. Available: https://doi.org/10.1145/2610384.2628058

[19] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proc. Symp. Principles of Programming Languages (POPL)*. ACM, 1995, pp. 49–61.

[20] V. Kanvar and U. P. Khedker, "Heap abstractions for static analysis," *ACM Comput. Surv.*, vol. 49, no. 2, pp. 1–47, 2016.

[21] J. Lerch, B. Hermann, E. Bodden, and M. Mezini, "FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases," in *Proc. Int. Symp. Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 98–108.

[22] Y. Xu, C. Zhang, and G. Pu, "Revisiting the combination of static analysis error traces and dynamic symbolic execution: A potential approach for true positive confirmation (registered report)," in *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA Companion '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 124–132. [Online]. Available: https://doi.org/10.1145/3713081.3731720

[23] N. A. of Sciences, Medicine, Policy, G. Affairs, B. on Research Data, Information, D. on Engineering, P. Sciences, C. on Applied, T. Statistics *et al.*, *Reproducibility and replicability in science*. National Academies Press, 2019.

[24] D. Beyer, "State of the art in software verification and witness validation: Sv-comp 2024," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 299–329.

[25] A. Murali, N. Mathews, M. Alfadel, M. Nagappan, and M. Xu, "Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623321

[26] C. Niddodi, S. Nagy, D. Marinov, and S. Mohan, "Topr: Enhanced static code pruning for fast and precise directed fuzzing," 2023. [Online]. Available: https://arxiv.org/abs/2309.09522

[27] G. Li, T. Yuan, J. Lu, L. Li, X. Zhang, X. Song, and K. Zhang, "Exposing vulnerable paths: Enhance static analysis with lightweight symbolic execution," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, 2021, pp. 441–451.

[28] F. Molina, P. Ponzio, N. Aguirre, and M. Frias, "Learning to prune infeasible paths in generalized symbolic execution," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, 2022, pp. 494–504.

[29] Q. Yi, Y. Yu, and G. Yang, "Compatible branch coverage driven symbolic execution for efficient bug finding," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3656443