Exposing Vulnerable Paths: Enhance Static Analysis with Lightweight Symbolic Execution

Guangwei Li[†‡], Ting Yuan[†‡], Jie Lu[†], Lian Li[†‡*], Xiaobin Zhang[§], Xu Song[§], Kejun Zhang[§]
[†] *SKL of Computer Architecture, ICT, CAS, Beijing, China*
[‡] *University of Chinese Academy of Sciences, China*
[§] *Huawei Technologies Co. Ltd, China*
[†]{ liguangwei, yuanting, lujie, lianli}@ict.ac.cn
[§]{zhangxiaobin2, songxu1, kejunzhang}@huawei.com

2021 28th Asia-Pacific Software Engineering Conference (APSEC) | 978-1-6654-3784-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/APSEC53868.2021.00051

***Abstract*—Static analysis tools, although widely adopted in industry, suffer from a high false positive rate. This paper aims to refine the results of static analysis tools, by automatically searching for a vulnerable path from given defect report. To realize this goal, we develop SATracer, a novel tool which integrates symbolic execution techniques with static analysis. SATracer selectively skips those program parts which can be consistently updated by static analysis, thus drastically improving performance. We have applied SATracer to a set of 21 real-world applications. Evaluation results show that SATracer can successfully remove 71.4% false alarms reported by a commercial static analysis tool in 10 hours, and confirmed 29 real use-after-free bugs and 895 real null-pointer-dereference bugs.**

***Index Terms*—static analysis, symbolic execution**

I. Introduction

Static analysis has been widely adopted in detecting software vulnerabilities. There are many commercial and open source static analysis tools available [1]–[3]. Those tools apply various approximation and abstraction techniques so as to analyze large programs and detect defects efficiently. Such approximations successfully scale static analysis tools to large real-world applications with millions of lines of code (LOC). However, they may also bring in many false positives.

We aim to refine the results of static analysis tools, by automatically searching for a vulnerable path from a defect report. A defect report commonly consists of a *source* as the starting point, a *sink* as the bug exposing statement, and a few key steps to trigger or help understand the root cause of the bug. Given such information, we further extract a step-by-step path which is highly likely to trigger the reported defect. Thus, a reported defect is regarded as a true bug if a corresponding vulnerable path is identified. Otherwise, it is likely to be a false positive.

Symbolic execution [4]–[6] is a promising technique to realize our goal. However, traditional symbolic execution techniques suffer from the path explosion problem. These techniques systematically inspect every program path. Although precise, they cannot scale to large real-world programs which often have billions of program paths to be explored.

Hence, we propose a lightweight solution which mixes symbolic execution together with static analysis. Our approach explores program paths in a similar fashion to symbolic execution. For efficiency, instead of systematically exploring all program paths, we start symbolic execution from the entry function of the source of a defect and selectively skip some program parts during symbolic execution. For those skipped parts, the symbolic execution state is consistently updated according to static analysis results. There are two key questions to be addressed:

- Which program parts can be skipped ?
  Ideally, we wish to skip program parts without loss of precision. In practise, alias is one of the main sources of imprecise analysis results. Hence, we skip exploring the callee function of a method invocation if it does not introduce suspicious aliases (i.e., resulting in one pointer with multiple incoming values).
- How to update symbolic state with static analysis?
  We design an abstract symbolic state in such a way that each pointer points to at most one target. To consistently update the symbolic state with static analysis, we maintain a mapping between heap objects during symbolic execution and abstract locations in static analysis: abstract location O uniquely pointed to by variable x is consistently mapped to l, where l is the heap location x referring to during symbolic execution. The symbolic state then can be updated by interpreting each instruction, or by static analysis results (Section III-B).

We develop SATracer (a static-analysis based symbolic execution tool), to automatically expose vulnerable paths from a defect report. Given a defect report, SATracer performs path exploration from the source of the defect report. To help speed up symbolic execution, a classic context- and flow-sensitive pointer analysis [3] is applied to the underlying program. We maintain an abstract symbolic state where each pointer must only point to at most one target. Those function call statements are skipped if their static analysis results follow the same property of the symbolic state. To the end, SATracer reports a confirmed bug if a vulnerable path from source to sink is identified within the given time frame.

We have evaluated SATracer using Juliet test suite [7], and an extensive set of 21 real-world applications. In our evaluation, we firstly generate a set of bug reports from a

*Corresponding author.

2640-0715/21/$31.00 ©2021 IEEE
DOI 10.1109/APSEC53868.2021.00051

commercial static tool, then apply SATRACER to search a bug trace for each bug report. Experimental results show that SATRACER can effectively filter 71.4% of false alarms in less than 10 hours. In addition, we manually confirmed 29/41 real use-after-free bugs and 895/939 real null-pointer-dereference bugs from bug traces searched by SATRACER. There are 2,055 likely false alarms[1] filtered by SATRACER.

The contribution of this paper is as follows.

- We propose a novel methodology which successfully integrates static analysis with symbolic execution techniques. We show how symbolic execution results can be consistently updated with static analysis.
- We implement a new tool SATRACER, to automatically extract a vulnerable program path from a defect report given by static analysis tools. SATRACER selectively skips those program parts which can be consistently updated by static analysis, thus drastically improve performance.
- We applied SATRACER to a set of 21 real-world applications. The tool is effective at finding 924 real bugs out of 3,858 defect reports, effectively filtering 2,055 false alarms within 10 hours. It can significantly improve precision of defect reports from static analysis tools: 95.21% null-pointer-dereference bug traces and 70.7% user-after-free bug traces are confirmed as real bugs.

The rest of the paper is organized as follows. Section II overviews our approach using an example. We formally describe technical details of SATRACER in Section III and evaluate it in Section IV. Section V reviews related paper, and Section VI concludes this paper.

II. MOTIVATION

A. An example

We informally describe SATRACER using the example in Figure 1(a). The example is simplified from lib/fts-cycle.c in Coreutils [8]. There are two main functions: `hash_add()` (lines 22 - 28), and `hash_insert()` (lines 6 - 21). The `hash_insert()` function tries to insert `entry` into the global variable `hash_table`: the function `get_table()` is invoked at line 7 and returns `hash_table` to pointer `table`. The loop from lines 9 -14 iterates through the table for an available slot. If it succeeds, `entry` is inserted into `hash_table` and stored to parameter `matched` (line 17). The function then returns `0` (line 18). Otherwise, it returns `1` (line 20) to indicate insertion failure.

Function `hash_add()` allocates a new memory object `in` (line 23) and invokes `hash_insert()` to insert it into `hash_table` (line 25). The object is freed if insertion fails (line 26), otherwise the object is returned via pointer `added`.

a) The bug report:

Definition 1. *A **defect report** is a triple <**Source, Sink, {Step}**>, where **Source** is the start point of the defect, **Sink** is the instruction exposing the defect, and **{Step}** is a set of key instructions triggering the defect.*

[1]We examined 100 of them and confirmed that they were false positives.

Static analysis tools may report a use-after-free bug <**23, 27, {26}**>, due to false alias between `in` and `added`. Line 23 is the source location where an object is allocated, the object is then freed at line 26 (as a key step in the bug report), and used at line 27 (as sink triggering the bug).

The false alarm arises due to false alias relation - after invoking function `hash_insert()`, both `in` and `added` may point to the object created at line 23. Hence, it looks like a use-after-free bug since the object is freed at line 26 and used at line 27. However, alias only holds if insertion succeeds and the function call returns `0`. As a result, `added` is `NULL` instead of pointing to the freed object and a bug cannot be triggered at line 27.

b) Challenges: This example, although conceptually simple, reveals key challenges for static analysis tools: alias and path-sensitivity. To avoid such false alarms, tools need to infer the precise relation that "`in` alias with `added` only if `err==0`". How to precisely track alias together with precise path conditions (i.e., path-sensitive alias analysis) remains an open research question.

Value-flow based approaches with path-sensitive extensions [9]–[11] can extract some path information (based on control dependencies) and mitigate the problem to a certain degree. However, it is very challenging, if not impossible, to precisely track all useful path conditions inter-procedurally. As in our example, tools need to analyze that the variable `ret` is set to a distinct value upon different alias relation, and propagate those information throughout the program. It can be much more complex for real-world applications, with millions of variables and alias relations. Frequently, compromises are made, which result in false positives or false negatives. As an illustration, we modify the condition "`(err != 0)`" in line 26 to its opposite condition "`(err == 0)`" (highlighted in Figure 1(a)), which will reveal a true use-after-free bug. Neither SVF (the state-of-the-art value-flow based tool) nor Infer (a static analysis tool from Facebook) can successfully report a bug in this case, suggesting unsound trade-offs and sacrificed ability in those tools.

B. The SATRACER approach.

We address the above challenges in two steps: firstly, we employ a conservative tool to report as many bugs as possible; secondly, we refine the reported results by precisely checking path conditions against each reported bug. This is realized in SATRACER by searching for a *bug trace* for each reported defect. A *bug trace* is a feasible program path highly likely to trigger a reported bug, as defined below:

Definition 2. *The **bug trace** for bug <**Source, Sink, {Step}**> is a feasible bug-triggering program path from **Source** to **Sink**, with some or none instructions in **{Step}** to trigger the buggy condition.*

Figure 2 overviews SATRACER, which comprises three phases as illustrated below.

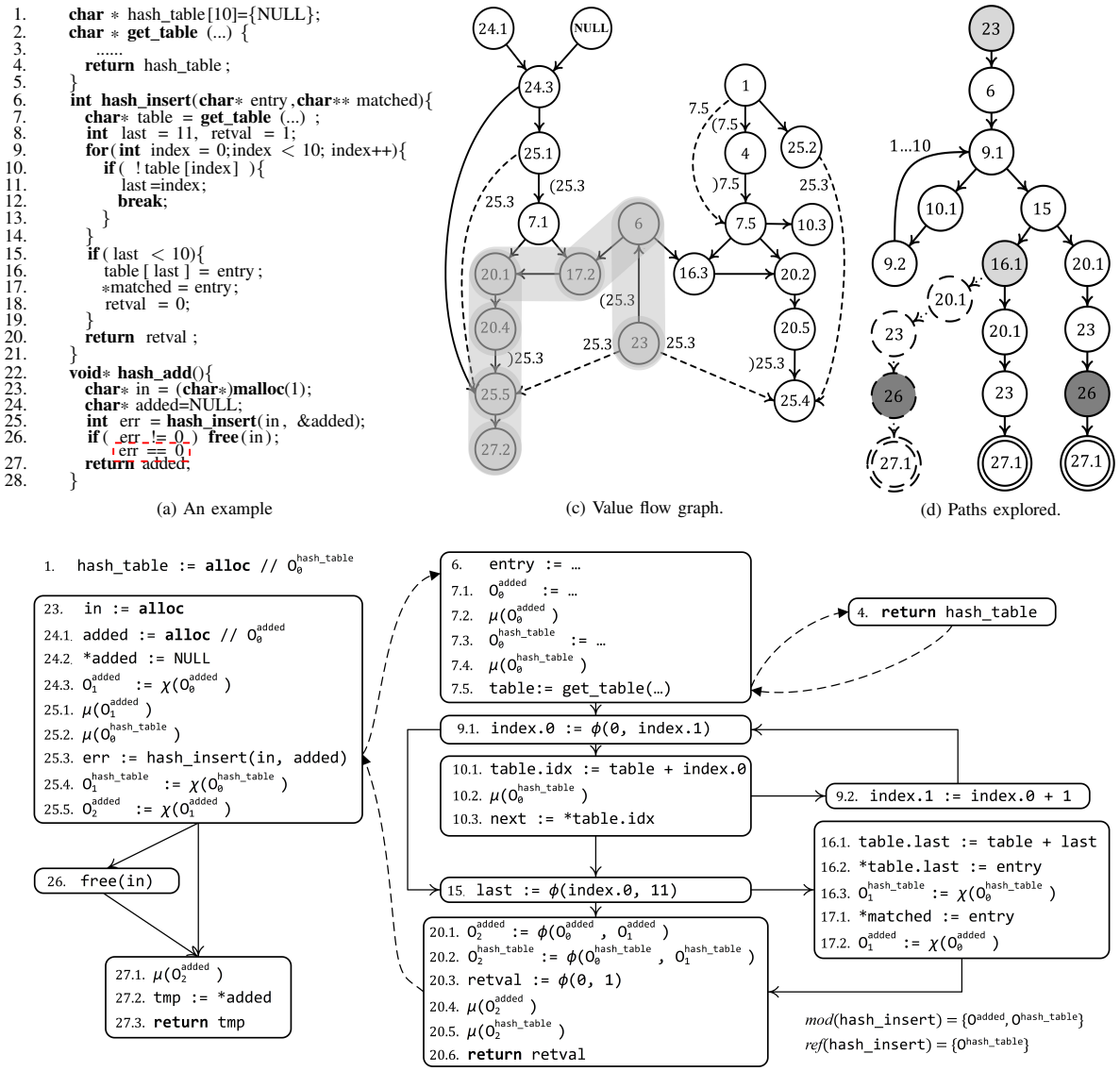


Fig. 1: A motivating example simplified from lib/fts-cycle.c in CoreUtils.

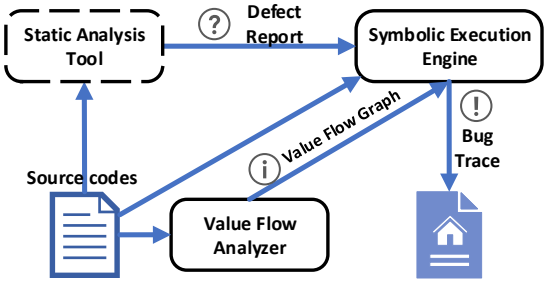


Fig. 2: Structure of SATRACER.

- **Step1: Static Analysis Tool** We leverage an off-the-shelf commercial tool for initial bug reporting. The tool reports a use-after-free bug <23, 27, {26}> for our motivating example in Figure 1(a).
- **Step2: Value-flow Pointer Analysis** We adopt existing value-flow based pointer analyses [12], [13] to build a flow- and context-sensitive value-flow graph for the program under analysis. The program is firstly translated into memory-SSA [13] (Section III-A1), then a context- and flow-sensitive value flow graph (Section III-A2) is constructed in a standard fashion. Figure 1(b) and Figure 1(c) give the memory-SSA representation and the value flow graph for our motivating example, respectively.

The value-flow graph captures def-use chains (both intra- and inter-procedurally) for all pointer variables in the program.

- **Step3: Path Exploration** This is the key step and is also the key contribution of this paper. From the source point, we explore for a vulnerable path to the sink. During path exploration, we maintain a symbolic program state. The symbolic program state is updated when executing an instruction (as in classical symbolic execution), or when skipping a function call. We selectively skips certain function calls by updating the symbolic state according to the value flow graph built in Step 2. Figure 1(d) depicts the program paths explored. If we modify the condition `"err != 0"` to an opposite condition `"err == 0"`, a bug trace will be identified as the path in gray lines.

In Step1, any static analysis tool can be plugged in SATRACER to refine its results. Hence, we will not discuss details in initial reporting.

III. METHODOLOGY

We implement SATRACER in LLVM [14] and adopt previous approaches [12], [13] in building the value-flow graph of a program. Path exploration is then performed with defect reports and the context- and flow-sensitive value-flow graph as inputs.

A. Value-flow Pointer Analysis

Following [3], [12], [13], [15], the value-flow graph is built upon a memory SSA (single static assignment) [16], [17] form as illustrated below.

TABLE I: types of instructions, where x, y, k, and z are variables, c is constant value, **op** represents binary operators (with unary operators being its special case), and L denotes the label for an instruction.

Name	Instruction
Allocation	x := **alloc**
Assign	$x := y \mid c$
Load	$x := *y$
Store	$*x := y$
Call	$x := F(...)$
Return	**return** y
Arithmetic	$x := y$ **op** z
Branch	**if** x **goto** L_t **else** L_f
PHI	$x := \phi(x1, x2, ...)$
MU	$\mu(x)$
CHI	$x := \chi(x)$

1) Memory SSA: Without loss of generality, we consider the set of LLVM instructions in Table I. The top half of the table are LLVM instructions in the standard SSA form. Note that here we do not distinguish objects allocated on stack or on heap (via **malloc**).

An insensitive Andersen-style pointer analysis [18] is firstly conducted as a pre-analysis to build SSA, as in [13]. The pre-analysis computes the following information: $pts(p)$ is the set of static memory locations pointed to by pointer p, $mod(F)$ and $ref(F)$ are the set of F's external memory locations modified and read by function F, respectively.

$$\text{[LOAD]}\quad \frac{x := *y \qquad O \in pts(y)}{\mu(O) \in [x := *y]^{\mu}}$$

$$\text{[STORE]}\quad \frac{*x := y \qquad O \in pts(y)}{O = \chi(O) \in [*x := y]_{\chi}}$$

$$\text{[CALL]}\quad \frac{x := F(...) \qquad O_m \in mod(F) \qquad O_r \in ref(F)}{\mu(O_r), \mu(O_m) \in [x := F(...)]^{\mu} \qquad O_m = \chi(O_m) \in [x := F(...)]_{\chi}}$$

$$\text{[REF]}\quad \frac{O_r \in ref(F)}{\mu(O_r)_{\in F} \in Entry_F^{\mu}}$$

$$\text{[MOD]}\quad \frac{O_m \in mod(F)}{\mu(O_m)_{\in F} \in Entry_F^{\mu} \qquad \mu(O_m)_{\in F} \in Exit_F^{\mu}}$$

Fig. 3: Rules to introduce μ and χ functions.

In memory SSA, we introduce two additional functions μ and χ, to indicate the potential implicit uses and defs of memory allocations for each instruction. For instruction S, we use S^{μ} and S_{χ} to denote the set of μ and χ instructions introduced for S, respectively. S^{μ} is inserted immediately before S, and S_{χ} is introduced immediately after S.

Figure 3 gives the standard rules to introduce μ and χ instructions. Given a load instruction x := *y, for each object $O \in pts(y)$, a $\mu(O)$ instruction is introduced (rule [LOAD]). Similarly, each store instruction $*x := y$ is annotated with a list of χ instructions, where an instruction $O := \chi(O)$ denotes an indirect update to object O pointed to by y (rule [STORE]).

At function call-sites, μ and χ instructions are introduced to propagate implicit uses and defs of memory allocations inter-procedurally. This is realized by the three rules: [CALL], [REF], and [MOD]. For each object $O_r \in ref(F)$, a $\mu(O_r)_{\in F}$ instruction is introduced at the entry of function F (rule [REF]), and a corresponding $\mu(O_r)$ instruction is introduced at its call-site $x := F(...)$ (rule [CALL]). For each object $O_m \in mod(F)$, a $\mu(O_m)_{\in F}$ function is introduced at both entry and exit of function F (rule [MOD]), and corresponding $\mu(O_m)$ and $O_m = \chi(O_m)$ instructions are inserted before and after its call-site $x := F(...)$ (rule [CALL]), respectively. Conceptually, the introduced μ and χ functions at function call-cites and function entry/exit act as extended parameters and return values of function F, to explicitly express inter-procedural indirect uses and defs of F, just like pass-by-value.

a) Memory-SSA example: After introducing μ and χ instructions, memory-SSA is then built using the standard SSA algorithm [19]. Figure 1(b) shows our example in memory-SSA. We label each instruction following its corresponding source line location, e.g., the statement at line 25 results in 5 instructions in memory SSA form, labeled from `25.1` to `25.5`. The instruction `24.2:*added := NULL` introduces "`24.3` : $O^{added} := \chi(O^{added})$", since O^{added} is pointed to by `added`. Note that SSA will rename the instruction to "$O_1^{added} := \chi(O_0^{added})$". Given that $mod(\texttt{hash\_insert}) = \{O^{added}, O^{hash\_table}\}$, $\mu(O^{added})$ and $\mu(O^{hash\_table})$ are introduced at the entry (7.1 - 7.4), exit (20.4 and 20.5), and

$$\frac{x := y}{x \hookleftarrow y}\ [\text{COPY}] \qquad \frac{x := \phi(y_0, y_1)}{x \hookleftarrow y_0 \qquad x \hookleftarrow y_1}\ [\text{PHI}]$$

$$\frac{x := *y \qquad \mu(O) \in [x := *y]^{\mu}}{x \hookleftarrow O}\ [\text{LOAD}]$$

$$\frac{*x := y \qquad O' := \chi(O) \in [*x := y]_{\chi}}{O' \hookleftarrow y \qquad O' \hookleftarrow O}\ [\text{STORE}]$$

$$\frac{l : x := F(..., q_j, ...)}{PAR_F(q_j) \overset{(_l}{\hookleftarrow} q_j \qquad x \overset{)_l}{\hookleftarrow} RET_F}\ [\text{CALL}]$$

$$\frac{l : x := F(...) \qquad \mu(O) \in [x := F]^{\mu} \qquad \mu(O)_{\in F} \in Entry^{\mu}_F}{O_{\in F} \overset{(_l}{\hookleftarrow} \mu(O) \qquad \mu(O) \hookleftarrow O}\ [\text{CALLMU}]$$

$$\frac{l : x := F(...) \qquad O' := \chi(O) \in [x := F]_{\chi} \qquad \mu(O)_{\in F} \in Exit^{\mu}_F}{O' \overset{)_l}{\hookleftarrow} \mu(O)_{\in F} \qquad \mu(O)_{\in F} \hookleftarrow O_{\in F}}\ [\text{CALLCHI}]$$

Fig. 4: Value flow rules.

call-sites of function `hash_insert` (25.1 and 25.2). Here we also introduce pseudo instructions `7.1` : $\text{O}^{\texttt{added}} = ...$ and `7.3` : $\text{O}^{\texttt{hash\_table}} = ...$ to make it a valid SSA form. In addition, `25.4` : $\text{O}^{\texttt{added}} := \chi(\text{O}^{\texttt{added}})$ and `25.5` : $\text{O}^{\texttt{hash\_table}} := \chi(\text{O}^{\texttt{hash\_table}})$ are inserted after the call to `hash_insert`.

2) Value Flow Graph: With the annotated μ and χ functions, the value flow graph is then built flow-sensitively in a standard fashion. The value flow edge $y \hookleftarrow x$ represents a dependence (i.e., a def-use relation) between x and y. In addition to the def-use chains provided by LLVM's SSA, value flow edges are also introduced for indirect def-uses, as shown in Figure 4.

Value flow edges are straight-forwardly introduced for direct assignments via rule [COPY] and rule [PHI]. In [LOAD], the implicit use of memory allocation O is connected to the load instruction. In [STORE], $O' \hookleftarrow O$ signifies a weak update to O and such edge can be eliminated if a strong update is possible (i.e., when the stored value uniquely points to a singleton object O). The rule [CALL] encodes the standard pass-by-value semantics, where $PAR_F(q_j)$ is $F's$ corresponding formal parameter for actual parameter q_j, and RET_F is $F's$ return value. Note that inter-procedural value flows are annotated with call-site labels for context-sensitivity.

Rule [CALLMU] and rule [CALLCHI] introduce indirect inter-procedural value flows into and returning from callee functions, respectively. At a function call-site $l : x := F(...)$, for each memory allocation O used in F, the two value flows $O_{\in F} \overset{(_l}{\hookleftarrow} \mu(O)$ and $\mu(O) \hookleftarrow O$ are introduced. As such, O is explicitly passed to $O_{\in F}$ via an extended parameter $\mu(O)$. Similarly, for each memory allocation O modified in F, the two value flows $O' \overset{)_l}{\hookleftarrow} \mu(O)_{\in F}$ and $\mu(O)_{\in F} \hookleftarrow O_{\in F}$ are introduced to return $O_{\in F}$ to its corresponding caller value O', via an extended return value $\mu(O)_{\in F}$. By definition, $O_{\in F}$ is an object modified in F ($\mu(O_{\in F}) \in Exit^{\mu}_F$). Here the labels $(_l$ and $)_l$ are annotated on inter-procedural value flow edges

$$\frac{l : x := F(..., q_j, ...) \qquad RET_F \leftsquigarrow PAR_F(q_j)}{x \overset{l}{\hookleftarrow} q_j}\ [\text{SUM}]$$

$$\frac{l : x := F(..., q_j, ...) \qquad O' \overset{)_l}{\hookleftarrow} \mu(O')_{\in F} \qquad O'_{\in F} \leftsquigarrow PAR_F(q_j)}{O' \overset{l}{\hookleftarrow} q_j}\ [\text{SUMPAR}]$$

$$\frac{l : x := F(..., q_j, ...) \qquad O_{\in F} \overset{(_l}{\hookleftarrow} \mu(O) \qquad RET_F \leftsquigarrow O_{\in F}}{x \overset{l}{\hookleftarrow} \mu(O)}\ [\text{SUMRET}]$$

$$\frac{l : x := F(...) \qquad O_{\in F} \overset{(_l}{\hookleftarrow} \mu(O) \qquad O' \overset{)_l}{\hookleftarrow} \mu(O')_{\in F} \qquad O'_{\in F} \leftsquigarrow O_{\in F}}{O' \overset{l}{\hookleftarrow} O}\ [\text{SUMEXD}]$$

Fig. 5: Summary rules.

for matched context-sensitivity at call-site l.

a) Value flow summaries: In our analysis, we introduce four more additional rules to summarize the value flows in callee functions. We use the notation $y \leftsquigarrow x$ to denote that there is a realizable path from x to y in the value flow graph, where matched context-sensitivity is expected. Figure 5 presents the summary rules.

Rule [SUM] summarizes a realizable value flow path from $F's$ parameter to its return value ($RET_F \leftsquigarrow PAR_F(q_j)$), by introducing a summary edge at its call-site, i.e, $x \overset{l}{\hookleftarrow} q_j$. Here the summary edge is marked with a call-site label l. The rest 3 rules [SUMPAR], [SUMRET], and [SUMEXD] handle indirect inter-procedural value flows in different cases. In a nut shell, [SUMPAR], [SUMRET], and [SUMEXD] summarize indirect value flows from parameter to indirect return value, from indirect input to return value, and from indirect input to indirect return value, respectively.

b) Value flow graph example: The value flow graph of our example is given in Figure 1(c). For illustration, We label each node with its corresponding instruction label, e.g., variable `in` at line 23 is represented as a node numbered 23. Let us study the value flow path `25.5` $\overset{)_{25.3}}{\hookleftarrow}$ `20.4` $\hookleftarrow$ `20.1` $\hookleftarrow$ `17.2` $\hookleftarrow$ `6` $\overset{(_{25.3}}{\hookleftarrow}$ `23`, as highlighted in Figure 1(c). Edge `6` $\overset{(_{25.3}}{\hookleftarrow}$ `23` is introduced at the call-site `25.3:err := hash_table(in, added)`, which pass parameter `23:in :=` **`alloc`** to function `hash_insert`'s formal parameter (`6:entry := ...`). Next, instruction `17.1:*matched:=entry` and `17.2:`$\text{O}_1^{\texttt{added}} := \chi(\text{O}_0^{\texttt{added}})$ generate the edge `17.2` $\hookleftarrow$ `6` (i.e., $\text{O}_1^{\texttt{added}} \hookleftarrow$ `entry`) by rule [STORE]. Edge `20.1` $\hookleftarrow$ `17.2` is introduced at instruction `20.1:`$\text{O}_2^{\texttt{added}} := \phi(\text{O}_0^{\texttt{added}}, \text{O}_1^{\texttt{added}})$ according to rule [PHI]. Finally, given that `25.5:`$\text{O}_2^{\texttt{added}} := \chi(\text{O}_1^{\texttt{added}}) \in [\texttt{25.3:...}]_{\chi}$, and `20.4:`$\mu(\text{O}_2^{\texttt{added}}) \in Exit^{\mu}_{\texttt{hash\_insert}}$, the two value flow edges `25.5` $\overset{)_{25.3}}{\hookleftarrow}$ `20.4` $\hookleftarrow$ `20.1` are introduced by rule [CHI].

According to the summary rule [SUMPAR] (Figure 5), the value flow path `25.5` $\overset{)_{25.3}}{\hookleftarrow}$ `20.4` $\hookleftarrow$ `20.1` $\hookleftarrow$ `17.2` $\hookleftarrow$

TABLE II: symbolic states. l is location, c is constant value, t is top-level variable in SSA, l_c is location l with offset c, and t^ς is a symbolic value introduced for variable t.

Heap	H	::=	$\overline{l \mapsto o}$
Objects	o	::=	$\overline{c \mapsto v}$
Stack	S	::=	$\overline{t \mapsto v}$
Values	v	::=	$\varsigma \mid l_c$
Symbolic expression	ς	::=	$c \mid t^\varsigma \mid \varsigma$ **op** ς
Constraint lists	Σ	::=	$\emptyset \mid \Sigma, \varsigma \mid \Sigma, \neg\varsigma$

$6 \overset{(25.3}{\hookleftarrow} 23$ is summarized as $25.5 \overset{25.3}{\hookleftarrow} 23$, indicating that `in` is stored to $O_2^{\texttt{added}}$ by the call instruction `25.3:err := hash_table(in, added).`

B. Path Exploration

Given a defect triple <**Source,Sink,{Step}**>, we perform path exploration from the **Source**. More precisely, from the entry of F_{Source} where F_{Source} is function of **Source**.

1) Symbolic States: Table II presents the symbolic state. We maintain a symbolic state for each distinct path. The symbolic state consists of a heap H, a stack S, and a constraint list Σ. The heap H maps memory locations to objects, and the stack S consists a set of local variables. Constraints Σ are introduced at conditional branches, when symbolically executing a path. For clarity of presentation, we regard stack and global objects as also managed by the heap H and do not distinguish local variables in different stack frames. Stack objects and local variables will be automatically reclaimed when returning from the stack frame.

An object o is a map from constant offsets to values. Values are symbolic expressions ς or locations l_c. A symbolic expression ς can be a constant value c, an introduced symbolic value t^ς, or a binary operation of two symbolic expressions. A location l_c denotes the location with offset c to memory location l. For simplicity, we do not model symbolic locations and offset to a memory location is always concretized.

2) Updating Rules: Figure 6 gives the standard semantic rules when symbolically executing a instruction. In [ALLOC], for the allocation instruction $x :=$ **alloc**, we create a new location in Heap H, and the value of x is set to l_0. The two rules [ASSIGN] and [ARITHMETIC] update the value of the resulting variable. Note ϕ instructions are evaluated by rule [ASSIGN], since its incoming value is uniquely identified during path exploration.

Rule [LOAD] and [STORE] perform standard heap lookups and updates, respectively. In [LOAD], an heap lookup for $x := *y$ is only performed if the location l_c (referred by y) is in heap H. The case where $c \notin H[l]$ is handled by rules with static analysis integration, as discussed in the following section. In rule [STORE], if $c \notin H[l]$, the location l_c is lazily initialized in H and updated.

In [CALL], the stack is updated by copying values of actual parameters to formal parameters of the callee function, and execution continues from the entry of the callee function. In [RETURN], the return value is returned to the caller and local variables ($y_F \in F$) and local stack objects ($l_F \in F$) are discarded. Finally, a branch instruction will fork two states, one for the true target (rule [BRANCHT]) and one for the false target (rule [BRANCHF]), to explore both targets. The constraint list is updated accordingly.

$$\frac{x := \textbf{alloc} \qquad l \notin dom(H) \qquad H' := H, l \mapsto o}{HS\Sigma \rightarrow H'S[x \mapsto l_0]\Sigma} \text{[ALLOC]}$$

$$\frac{x := y}{HS\Sigma \rightarrow HS[x \mapsto S[y]]\Sigma} \text{[ASSIGN]} \qquad \frac{x := y \textbf{ op } z}{HS\Sigma \rightarrow HS[x \mapsto S[y] \textbf{ op } S[z]]\Sigma} \text{[ARITHMETIC]}$$

$$\frac{x := *y \qquad y \mapsto l_c \qquad c \in dom(H[l])}{HS\Sigma \rightarrow HS[x \mapsto H[l][c]]\Sigma} \text{[LOAD]}$$

$$\frac{*x := y \qquad x \mapsto l_c \qquad H[l]' := H[l] \setminus c, c \mapsto S[y]}{HS\Sigma \rightarrow H[l \mapsto H[l]']S\Sigma} \text{[STORE]}$$

$$\frac{x := F(..., k, ...)}{HS\Sigma \rightarrow HS, PAR_F(k) \mapsto S[k]\Sigma} \text{[CALL]}$$

$$\frac{x := F(...) \qquad l_F \in F \qquad y_F \in F \qquad H' := H \setminus l_F \qquad S' := S \setminus y_F}{HS\Sigma \rightarrow H'S'[x \mapsto S[REF_F]]\Sigma} \text{[RETURN]}$$

$$\frac{\textbf{if } x \textbf{ goto } L_t ... \qquad x \mapsto \varsigma}{HS\Sigma \rightarrow HS\Sigma, \varsigma} \text{[BRANCHT]} \qquad \frac{\textbf{if } x \textbf{ ... else } L_f \qquad x \mapsto \varsigma}{HS\Sigma \rightarrow HS\Sigma, \neg\varsigma} \text{[BRANCHF]}$$

Fig. 6: Symbolic execution semantics.

3) Static analysis integration: Figure 7 gives the rules. Rule [INIT] initializes symbolic states by assigning a value to each parameter of the entry function F_{Source}. A new label l is assigned to each pointer parameter, and a symbolic value is given to each non-pointer parameter.

Rule 1. *If variable x points to abstract location O only, then O maps to a unique location of value x.*

In addition to the symbolic state in Table II, we maintain an additional map $M := \overline{O \mapsto l}$, which maps static analysis object O to a unique location l in symbolic heap H. Following Rule 1, the deduction rules [MAP] and [CHI] updates the map from direct assignments, and from strong updates (i.e., $\{O' := \chi(O)\} == \{[*x := y]_\chi\}$ and $O' \not\leftarrow O$), respectively.

We apply lazy initialization [20], [21] when the instruction $x := *y$ loads from a new location not in the heap. There are 3 different cases: 1) rule [LOADVAL] handles the case when x is a non-pointer value, heap H is lazily initialized with the new location and x is given a symbolic value; 2) in rule [LOADPTR], x is a pointer value pointing to an untracked object O, heap H is lazily initialized a new label l' pointing to an empty object; 3) in rule [LOADPTRM], x is a pointer value pointing to a tracked object O (i.e., $O \in dom(M)$, x is assigned the tracked value of O, i.e,. H[M[O]].

Rule 2. *For any value k returned from function call $x := F(...)$, i.e, x and $O' := \chi(O) \in [x := F(...)]_\chi$, we skip symbolically executing the function call if there is only one summary value flow edge to k. The symbolic state is then consistently updated by value flow summaries.*

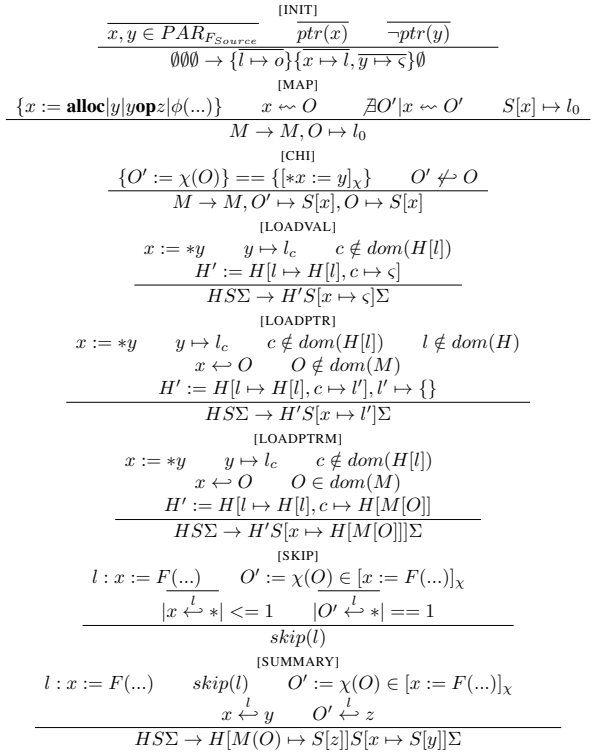


Fig. 7: Symbolic execution integrated with static analysis.

According to Rule 2, we skip symbolically executing a function call if it does not introduce spurious alias, i.e., there exists only one incoming value flow edge to all values (more precisely, to values related to the reported defects) returned (directly or indirectly) from the function call. Rule [SKIP] handles this particular case. The skipped function call is updated by rule [SUMMARY], where the return value can be consistently updated by its unique incoming value. Note that in this rule, we do not elaborate the detailed cases when the incoming value is a static object O. In that case, the return value is similarly updated as rule [LOADPTR] and [LOADPTRM].

C. SATracer

Figure 1(d) summarizes the entire process and illustrates the explored paths for our motivating example. For clarity, we do not list all traversed instructions, and each node is labeled with the first instruction of a traversed basic block in Figure 1(b).

From the start, we have the initial state $\emptyset\emptyset\emptyset$. The instruction `24.3:`$O_1^{\texttt{added}} := \chi(O_0^{\texttt{added}})$ strong updates $O_1^{\texttt{added}}$. Hence, we can infer that $O_1^{\texttt{added}}$ has same value as `added`. The instruction `25.3:err := hash_insert(in,added)` introduce two summary edges to `25.5:`$O_2^{\texttt{added}} := \chi(O_1^{\texttt{added}})$, i.e., 25.5 $\overset{25.3}{\hookleftarrow}$ 23 and 25.5 $\overset{25.3}{\hookleftarrow}$ 25.1. Thus, the function call cannot be skipped and we symbolically execute the callee function from `6:entry := ...`. At instruction `7.5:table := get_table(...)` , the return value `table` has one unique incoming value 7.5 $\overset{7.5}{\hookleftarrow}$ 1 and the function call is skipped. Consequently, `table` is updated by `hash_table`.

In this example, there is no bug trace for the defect report <23,27.3,{26}>. However, if we modify the error condition "`(err !=0)`" to "`(err == 0)`" (line 26 in Figure 1(a)), a bug trace is identified as in dashed lines (distinct parts highlighted).

D. Implementation

We adopt traditional optimizations and trade-offs for symbolic execution in SATracer, as summarized below:

- *Slicing.* Given a defect report <**Source, Sink, {Step}**>, we slice the program by eliminating instructions without transitive data- or control-dependences to **Source, Sink,** and **{Step}**. This optimization reduces memory usage by 13.05%, and improves runtime performance by 9.03%.
- *Path traversal heuristics.* We implement different path exploration strategies, including the default strategy which always explores the false target first, and a distance-guided heuristics which makes decisions based on the distance (in number of block) to sink.
- *loop Unrolling.* Loops are simply handled by unrolling a loop for at most 3 times.
- *Constraint simplification.* We only consider constraints introduced by common arithmetic and comparison operations, complex operations such as bit-level operations are omitted for better performance.
- *Bounding Computational Resources.* We bound the time of SATracer in tracing a defect report. By default, it is 10 secs. Furthermore, the maximal search depth is limited to 500 basic blocks to avoid path explosion.

IV. Evaluation

We evaluate the effectiveness and efficiency of SATracer by applying it to trace use-after-free and null-pointer-dereference bugs reported by a static analysis tool. Two set of benchmarks are used in our evaluation: 1) the Juliet test suite, and 2) a set of 21 real-world open source applications (Table IV). We compare the results of SATracer with ClangSA, the Clang static analyzer. All our experiments are conducted on a platform with 2 Intel Xeon Gold 6230@2.10GHz CPUs and 512GB memory.

A. Juliet Test Suite

TABLE III: Experiment results on the Juliet test suite.

#CWEID	#Testcases[2]	SATracer			ClangSA	
		#Potential	#TP	#FP	#TP	#FP
CWE416	393	395	388	0	18	0
CWE476	288	149	148	0	151	0
Total	681	544	536	0	169	0

The Juliet test suite consists of a large collection of different types of small defect programs (each defect program

[2]Without Windows specific and null-check-after-use cases.

is often accompanied with a correctly fixed version), and is widely used in evaluating bug detection tools. We evaluated SATRACER using the 2 defect types: CWE 416 (use-after-free), and CWE476 (null-pointer-dereference).

Table III summarizes the results. In total, there are 681 test cases (Column 2) and the static analysis tool generates a total of 544 defect reports (Column 3), including 395 use-after-free and 149 null pointer dereference bugs. Among the 544 reports, SATRACER identified a bug trace for 536 reports (Column 4), successfully removing 8 false alarms. Overall, SATRACER reports 388 use-after-free bugs and 148 null-pointer-dereference bugs, with no false positives. In comparison, ClangSA reports much less use-after-free bugs (18) and more null-pointer-dereference bugs (151), with no false positives. The reason that ClangSA is able to report more null-pointer-dereference bugs is because the underlying commercial tool reports a null-pointer-dereference bug only if the tool identifies a propagation path from the constant null value to a variable being dereferenced. On the other hand, ClangSA adopts a heuristics and regards a variable as a potential null value if this variable has been null checked.

B. Real-world Applications

Table IV reports the results over a set of 21 real-world applications. Sizes of those applications range from 5 KLOC (wrk) to 309 KLOC (httpd), with a total of 1,666 KLoc (Column 2). The static analysis tool reported 904 use-after-free bugs (Column 3) and 2,954 null-pointer-dereference bugs (Column 8) in the 21 applications. From this initial set of potential bugs, we report the number of true bug traces (TP), false bug traces (FP), memory usages, and analysis times of SATRACER. The analysis time includes the time in building the context- and flow-sensitive value flow graph, as well as the time in symbolically exploring a bug trace. In this experiment, we trace each defect report with a time limit of 10 seconds by default, and give each benchmark a time budget of 2 hours. For each benchmark, we run SATRACER for at least 3 times and report the average data over the 3 runs.

Overall, SATRACER successfully identified 29 real use-after-free (Column 4) and 895 real null-pointer-dereference (Column 9) bug traces, effectively removing 71.4% of the false alarms from the static analysis tool. We examined 100 reports without bug trace and confirmed that they were all false positives. There are also 12 false use-after-free (Column 4) and 45 false null-pointer-dereference (Column 9) bug traces, with an overall false positive rate of 5.88% (29.3% for use-after-free bugs and 4.79% for null-pointer-dereference bugs). Those 57 false positives are due to unresolved path conditions (e.g., bitwise operations, library calls, and pointer operations) and complex data structures (e.g., red-black tree and double linklist). The memory consumption (Column 5 and 10) and time cost (Column 6 and 11) in Table IV present the performance of SATRACER on each application.

Compared to ClangSA, SATRACER reports much more real bugs, including 22 more use-after-free bugs (Columns 4 and 7) and 861 more null-pointer-dereferences bugs (Columns 9 and 12), with a much lower false positive rate (5.88% of SATRACER vs 84%of ClangSA).

1) Implementation Trade-offs: We discuss and evaluate different implementation trade-offs in SATRACER. We compare different trade-offs in terms of their impact to efficiency and effectiveness, reflected as the percentage of timeouts and the precision of bug traces.

Figure 8(a) and Figure 8(b) compare the results under different time limits in tracing a potential use-after-free or null-pointer-dereference bug, respectively. As time limits increase (from 5s to 100s), the number of timeouts drops noticeably, with a steady increase on number of identified traces and number of confirmed false positives. The false positive rates of bug traces only vary slightly.

Figure 8(c) and Figure 8(d) study different design trace offs. The 4 configurations differ from **default** as follows: **priority** uses a different path traversal strategy which favors a branch target with shorter distance to sink, **no-slice** does not perform slicing, **no-step-in** always updates function call results with static analysis summaries, **stepin** performs symbolic execution without skipping function calls.

There are only slight differences between **priority** to **default**. Compare **default** with **no-slice**. The number of timeouts in **no-slice** is noticeably larger, increasing from 13.85% to 19.26% for null-poiter-dereference bugs. Overall, the slice optimization can effective reduce memory usages by 13.05%, and improve runtime performance by 9.3%. Finally, in the **no-step-in** configuration, the precision drops significantly, due to imprecise static analysis summaries. Compare **default** to **step-in**, the number of timeouts increases significantly, from 45.8% to 63.4% for use-after-free bugs (Figure 8(c)), with very slight differences on bug trace precision. This suggests that SATRACER can significantly improve symbolic execution performance with summarized static analysis information, while preserving same level of precision.

C. Case Study In Real-World Applications

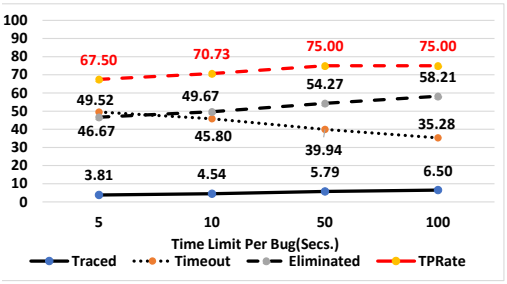
We manually examined all 951 bug traces produced by SATRACER, to confirm whether it is a real bug or a false positive. Two studies, one true positive and one false positive, are presented in this section for detailed discussion.

1) A true positive case: Figure 9 presents a new use-after-free bug from CoreUtils (lib/fts.c) which has never been reported before. For clarity, we only show the key bug-triggering steps with many details (e.g., path condition variables) skipped. To help understand the bug triggering path, we highlight relevant value flows in the graph, with solid lines for direct value flows, and dashed lines for indirect value flows.

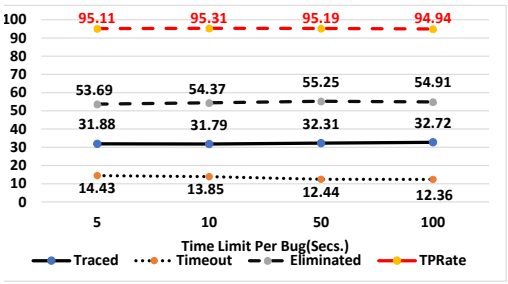
The bug is triggered as follows. First, the function `fts_alloc` is invoked at line 1475, which returns a newly created memory object (line 1923) with field `p->fts_path` set by `sp->fts_path`, i.e., the two fields point to a same object. Next, at line 1481, the conditional test checks for return value of `fts_palloc`. In function `fts_palloc`, line 1981 frees `sp->fts_path` via `realloc` then immediate reset it at line 1987. Finally, although `sp->ftp_path` is reset, the

TABLE IV: Experiment results on 21 real-world applications.

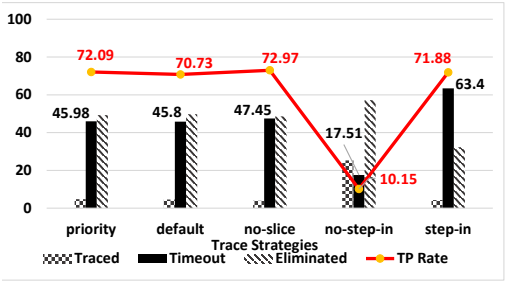
Project	Lines (Klocs)	USE_AFTER_FREE					NULL_POINTER_DEREFERENCE				
		SATracer				ClangSA	SATracer				ClangSA
		Pt.	TP./FP.	Memory (MB)	Time (sec.)	TP./FP.	Pt.	TP./FP.	Memory (MB)	Time (sec.)	TP./FP.
wrk	5	0	0/0	102	13.63	0/0	30	3/3	115	14.53	0/3
bzip2	6	1	0/0	93	14.18	0/0	4	0/0	1,422	522.49	0/0
memcached	12	4	0/0	326	23.04	0/1	95	23/2	437	302.37	0/9
mujs	15	1	1/0	2,532	32.10	0/0	32	0/0	5,716	282.35	0/0
less	20	29	0/2	718	84.71	0/2	78	13/0	18,808	82.16	1/0
icecast	21	60	5/0	663	192.28	0/0	403	278/0	1,555	392.44	0/2
shadowsocks	26	7	1/0	1,726	385.32	0/21	66	1/2	1,679	102.18	0/19
darknet	29	83	0/4	953	504.69	0/0	563	264/0	1,077	312.41	4/2
sed	33	10	4/0	123	65.99	3/1	1	0/0	134	42.11	0/3
gzip	43	1	0/0	973	1014.49	0/1	1	0/0	294	52.10	0/0
make	44	54	0/0	3,523	1105.89	0/1	12	0/0	1,736	796.84	0/11
goaccess	51	82	1/2	664	222.38	4/1	101	11/1	523	72.15	0/15
grep	81	17	3/0	248	86.92	0/0	48	18/0	467	196.89	9/10
tar	83	90	5/1	2,114	645.91	0/1	214	6/6	1,450	907.19	0/17
sendmail	95	25	0/0	4,320	193.76	0/0	37	1/2	3,505	566.118	0/21
bison	104	34	0/0	1,535	904.84	0/0	43	0/4	1,502	312.39	0/12
bash	113	200	0/0	4,965	7199.01	0/1	337	26/1	35,322	2724.76	14/36
curl	145	17	1/2	3,193	512.63	0/0	22	0/2	12,083	692.76	0/8
gnugo	209	10	0/0	5,330	232.28	0/0	140	81/0	6,066	482.53	0/5
coreutils	222	171	5/0	6,619	1623.60	0/0	267	12/0	3,227	1763.72	0/25
httpd	309	8	3/1	563	102.19	0/1	460	158/16	30,368	5407.67	6/29
summary	1666	904	29/12	-	4.21(h)	7/31	2954	895/45	-	4.45(h)	34/227



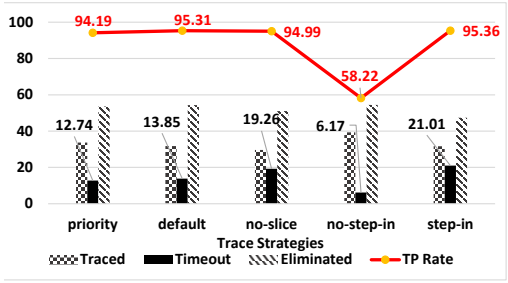
(a) Results with different time-limits (UAF)



(b) Results with different time-limits (NPD)



(c) Results with different implementation trade-offs (UAF)



(d) Results with different implementation trade-offs (NPD)

Fig. 8: Results with different implementation trade-offs.

field `p->fts_path` still points to the freed object. Following the two-headed arrow lines, *p* is returned to its caller and any successive access to `p->fts_path` triggers an error.

2) A false positive case: Figure 10 gives a false bug trace extracted from tar-1.32/lib/wordsplit.c. The sub-path ① ⇝ ② ⇝ ③ is highlighted in the bug trace. First, in the first iterator of the loop, `p` is assigned the value given

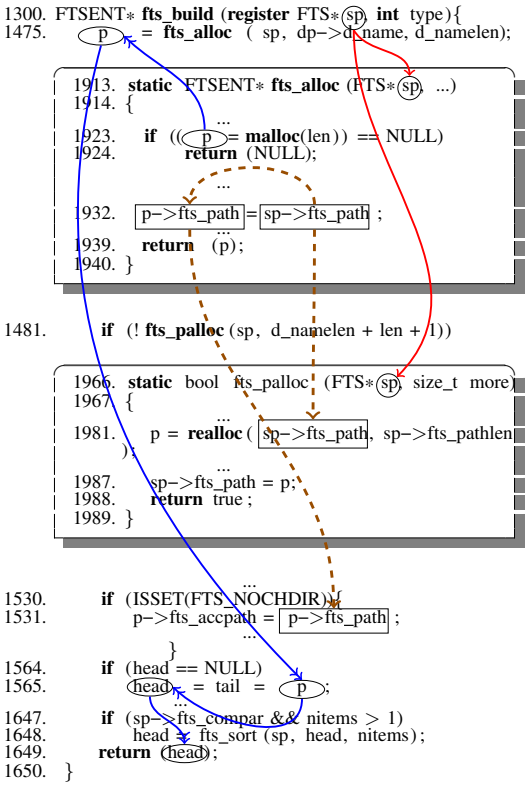


Fig. 9: A true positive bug trace from CoreUtils-8.32.

by `wsp->wsp_head` (①), then it is freed at line 1504. As a result `wsp->wsp_head` may point to the already freed buffer. Then, in the next loop iterations, if the condition test at line 463 always holds (① ⇝ ② ⇝ ③), the highlighted block (④) is never executed to reset `wsp->wsp_head`, and it remains to point to the freed buffer. Finally, after the loop terminates, `SATracer` report a bug trace revealing the dangling pointer `wsp->wsp_head`.

However, the bug trace is not feasible. `wordsplit`, a data structure is designed in such a way that the `prev` field of the head node `wsp->wsp_head` is `NULL`. Hence, in this case, when removing from the head of the list, we will always take false branch ④, which will reset `wsp->wsp_head` to a new header whose `prev` is set to `NULL`. This false positive is due to the limitation of SATracer in modeling input conditions. The initial introduced symbolic input values cannot preserve all invariants followed by the system, especially for data structures such as linked lists and trees.

V. Related Works

Infer [2] verifies selected properties of a target program by locally reasoning inference questions in the form of Hoare triples based on separation logic [22] and bi-abduction [23]. CBMC [24], a bounded model checker, extends model checking techniques to verify memory safety properties. Clang Static Analyzer [1] performs static symbolic execution within a

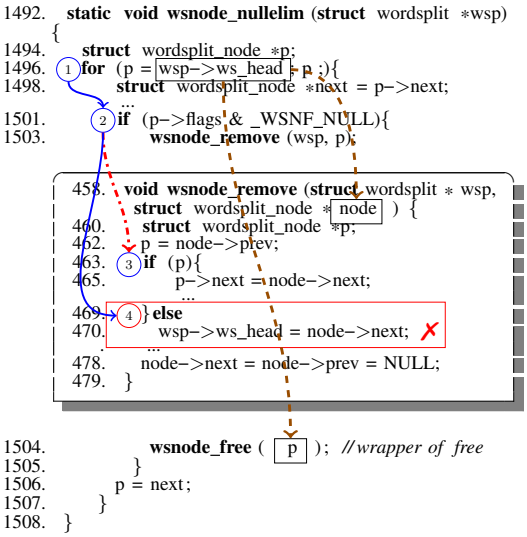


Fig. 10: A false positive bug trace from tar-1.32.

file to detect various bug types including use-after-free and null-pointer-dereference. FastCheck [9], Saber [13], [25], and Pinpoint [26] extend value-flow based pointer analyses with path constraints to detect deep inter-procedural bugs involving complex aliases. For efficiency, those tools apply various heuristic approaches together with formal analyses to analyze large real-world applications.

Symbolic execution tools like EXE [27], KLEE [28] and Cloud9 [29] have been used in a variety of domains, including software testing, bug detection, security and software and protocol verification. There have been several approaches trying to optimize symbolic execution with static program analysis. Most of those approaches leverage static analysis information to provide guidance for more effective path exploration. For instance, Mayhem [30] static analyzes program paths containing symbolic address/pointer operations, as a guidance for path exploration since those are more likely to be exploited. AEG [31] uses heuristics that prioritize bug paths and the work [32] proposes to favor program path satisfying a certain property, such as memory safety. Directed symbolic execution [33] tries to solve the line reachability (i.e, trigger a specified line) problem with guided path exploring strategies. WOODPECKER [34], a rule-based directed symbolic execution tool, applies path slicing to prune redundant paths irrelevant to given rules.

VI. Conclusion

We propose a novel symbolic execution approach integrated with static analysis, to automatically expose a vulnerable program path from a defect report. We develop SATracer and evaluate it using a set of 21 real-world applications. Evaluation results show that the approach is able to significantly improve symbolic execution performance by consistently updating symbolical results using static analysis information, while preserve similar precision.

REFERENCES

[1] "Clang static analyzer," 2020.

[2] "Facebook infer," 2020.

[3] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, pp. 265–266, 2016.

[4] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[5] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[6] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[7] "Juliet test suite for c/c++," 2017.

[8] "Gnu core utilities," 2020.

[9] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 480–491, 2007.

[10] Y. Sui, S. Ye, J. Xue, and P.-C. Yew, "Spas: Scalable path-sensitive pointer analysis on full-sparse ssa," in *Asian Symposium on Programming Languages and Systems*, pp. 155–171, Springer, 2011.

[11] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "Smoke: scalable path-sensitive memory leak detection for millions of lines of code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 72–82, IEEE, 2019.

[12] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pp. 460–473, 2016.

[13] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Transactions on Software Engineering*, vol. 40, no. 2, pp. 107–122, 2014.

[14] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.

[15] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 289–298, IEEE, 2011.

[16] F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich, "Effective representation of aliases and indirect memory operations in ssa form," in *International Conference on Compiler Construction*, pp. 253–267, Springer, 1996.

[17] D. Novillo *et al.*, "Memory ssa-a unified approach for sparsely representing memory operations," in *Proc of the GCC Developers' Summit*, Citeseer, 2007.

[18] L. O. Andersen, *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[20] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 553–568, Springer, 2003.

[21] X. Deng, J. Lee, *et al.*, "Efficient and formal generalized symbolic execution," *Automated Software Engineering*, vol. 19, no. 3, pp. 233–301, 2012.

[22] P. O'Hearn, J. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *International Workshop on Computer Science Logic*, pp. 1–19, Springer, 2001.

[23] C. Calcagno, D. Distefano, P. W. O'hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," *Journal of the ACM (JACM)*, vol. 58, no. 6, pp. 1–66, 2011.

[24] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 389–391, Springer, 2014.

[25] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 254–264, 2012.

[26] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang, "Pinpoint: Fast and precise sparse value flow analysis for million lines of code," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 693–706, 2018.

[27] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.

[28] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.

[29] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the sixth conference on Computer systems*, pp. 183–198, 2011.

[30] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, pp. 380–394, IEEE, 2012.

[31] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[32] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu, "Regular property guided dynamic symbolic execution," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 643–653, IEEE, 2015.

[33] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *International Static Analysis Symposium*, pp. 95–111, Springer, 2011.

[34] H. Cui, G. Hu, J. Wu, and J. Yang, "Verifying systems rules using rule-directed symbolic execution," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 329–342, 2013.