Fast and Precise Static Null Exception Analysis With Synergistic Preprocessing

Yi Sun, Chengpeng Wang, Gang Fan, Qingkai Shi, *Member, IEEE*, and Xiangyu Zhang, *Member, IEEE*

***Abstract*—Pointer operations are common in programs written in modern programming languages such as C/C++ and Java. While widely used, pointer operations often suffer from bugs like null pointer exceptions that make software systems vulnerable and unstable. However, precisely verifying the absence of null pointer exceptions is notoriously slow as we need to inspect a huge number of pointer-dereferencing operations one by one via expensive techniques like SMT solving. We observe that, among all pointer-dereferencing operations in a program, a large number can be proven to be safe by lightweight preprocessing. Thus, we can avoid employing costly techniques to verify their nullity. The impacts of lightweight preprocessing techniques are significantly less studied and ignored by recent works. In this paper, we propose a new technique, BONA, which leverages the synergistic effects of two classic preprocessing analyses. The synergistic effects between the two preprocessing analyses allow us to recognize a lot more safe pointer operations before a follow-up costly nullity verification, thus improving the scalability of the whole null exception analysis. We have implemented our synergistic preprocessing procedure in two state-of-the-art static analyzers, KLEE and Pinpoint. The evaluation results demonstrate that BONA itself is fast and can finish in a few seconds for programs that KLEE and Pinpoint may require several minutes or even hours to analyze. Compared to the vanilla versions of KLEE and Pinpoint, BONA respectively enables them to achieve up to 1.6x and 6.6x speedup (1.2x and 3.8x on average) with less than 0.5% overhead. Such a speedup is significant enough as it allows KLEE and Pinpoint to check more pointer-dereferencing operations in a given time budget and, thus, discover over a dozen previously unknown null pointer exceptions in open-source projects.**

***Index Terms*—Null exception analysis, static analysis, dataflow analysis, symbolic execution, and path sensitivity.**

I. Introduction

NULL pointer exception (NPE), listed as the CWE top 25 most dangerous software weaknesses in past five years [1], occurs when a program dereferences a memory pointer that is expected to be valid but is “null”. NPE has been one of the most common software vulnerabilities due to the wide use of pointer-dereferencing operations in the code written in modern programming languages. According to our investigation and as shown in Fig. 1, C programs in SPEC CPU 2006 [2], a standard and widely-used benchmark suite, contain up to 64% pointer instructions, in which up to 61% dereference pointers. Thus, it is highly challenging to guarantee the absence of NPE even if advanced quality assurance techniques are used. It has been reported that over 70 NPE vulnerabilities are collected by the CVE database every year [3] and as of July 2024, there are over 3000 CVEs involving NPEs [4]. These NPE vulnerabilities often lead to Denial of Service attacks, causing immeasurable losses.

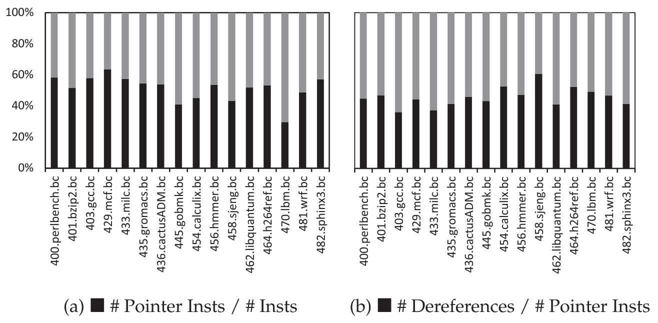


Fig. 1. Percentage of pointer operations and pointer dereferences. The X-axes list 18 programs from SPEC2006.

To prevent NPE as well as its serious consequences, researchers have proposed a large number of static dataflow analyses in the past decades, including general approaches able to detect various bug types [5], [6], [7], [8], [9] and techniques specially designed for detecting NPEs [10], [11], [12], [13]. We appreciate these existing works and acknowledge their contributions. However, it is hard to say the problem of NPE detection has been completely addressed, given the increasing number of NPEs every year. A notable problem of static analysis is that, when high precision like path sensitivity is required, it often takes many hours to complete the analysis due to the heavy use of costly techniques such as symbolic execution and SMT solving. To make these techniques scalable for large codebases, they either make unsound assumptions that sacrifice precision and recall, or seek some advanced techniques, such as sparse dataflow analysis [6], [14], [15], parallel static analysis

Received 6 February 2024; revised 20 July 2024; accepted 17 September 2024. Date of publication 23 September 2024; date of current version 14 November 2024. Recommended for acceptance by S. Ryu. *(Corresponding author: Qingkai Shi.)*

Yi Sun, Chengpeng Wang, and Xiangyu Zhang are with Purdue University, West Lafayette, IN 47907 USA (e-mail: sun624@purdue.edu; wang6590@purdue.edu; xyzhang@purdue.edu).

Gang Fan is with Hong Kong University of Science and Technology, Hong Kong, People’s Republic of China (e-mail: gfan@cse.ust.hk).

Qingkai Shi is with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, People’s Republic of China (e-mail: qingkaishi@nju.edu.cn).

Digital Object Identifier 10.1109/TSE.2024.3466551

[16], [17], [18], and specially designed type systems [10], to name just a few.

Different from the aforementioned techniques that focus on the dataflow analysis itself, in this paper, we advocate a design that is less studied and significantly underestimated by previous works. In the design, one or more lightweight but sound preprocessing procedures are conducted before the main static analysis. In the application scenario of detecting NPEs, such preprocessing procedures are capable of quickly identifying a large number of safe pointer-dereferencing operations so that we will not need to take a costly technique, e.g., symbolic execution and SMT solving, to verify their safety. Therefore, the performance of the whole program analysis is improved. For instance, unification-based flow- and context-insensitive alias analysis [19] is commonly used as a preprocessing procedure, e.g., in [20], [21], because the preprocessing alias analysis is of almost linear complexity and can answer alias queries in constant time. Using it as a preprocessing procedure, we can easily identify pointers that may be null and must not be null, thus avoiding expensive checks on those non-null pointers in the follow-up NPE analysis.

However, we observe that all existing approaches, e.g., [20], [21], independently utilize preprocessing procedures and fail to deeply explore their potential. Our key insight is that different preprocessing procedures can promote each other, exhibiting a synergistic effect. For NPE detection, the synergistic effects let us identify a lot more non-null pointers than using multiple preprocessing procedures independently. While the idea of utilizing mutually beneficial static analyses was studied in general settings particularly for compiler optimization [22], [23], [24], [25], [26], it has not been explored in the context of scaling precise bug or NPE detectors via preprocessing. More specifically, existing approaches cannot reply to the questions about what and how specific static analyses can promote each other for NPE detection. This paper provides an answer for NPE detection.

In this work, for detecting null pointer exceptions, we identify two lightweight and sound preprocessing techniques, which we refer to as the global value-flow analysis (VFA) and the local null-check analysis (NCA), and study the synergistic effects between them. VFA propagates the dataflow facts of whether a pointer variable is null across the function borders. NCA performs dataflow analysis to discover if a pointer variable v, albeit may be null, is guarded with a null-check statement such as `if (v != null)` or `*v = u`.[1] VFA can be boosted by NCA in the sense that NCA provides more null or non-null dataflow facts for VFA to propagate. NCA can be boosted by VFA in the sense that, since null or non-null dataflow facts are propagated via VFA to other program variables, we can identify more null-check statements via NCA. Such mutual promotion eases the burden of subsequent expensive static analysis and, thus, dramatically improves the analysis performance. We provide a detailed example in Section II to illustrate the idea.

On top of the LLVM compiler infrastructure [27], we have implemented the synergistic preprocessing technique, namely BONA, as a **B**ooster **O**f **N**ull **A**nalysis. We apply BONA to two precise static analyses for detecting null pointer exceptions. One is KLEE [9], a symbolic execution engine, and the other is Pinpoint [6], an industrial-strength bug detector with the precision of interprocedural path sensitivity. The evaluation is conducted based on the original benchmark programs of KLEE and Pinpoint in their papers. The experimental results demonstrated that BONA is very fast and can finish the synergistic preprocessing in a few seconds. Compared to the vanilla versions of KLEE and Pinpoint, BONA respectively enables them to achieve up to 1.6x and 6.6x speedup (1.2x and 3.8x on average) with less than 0.5% overhead. Such a speedup is significant enough as it allows KLEE and Pinpoint to check more pointer-dereferencing operations in a given time budget, leading to the discovery of a dozen previously unknown NPEs in open-source projects. We make the following three contributions in this paper:

- We identify two fast and sound preprocessing techniques and, for the first time to our best knowledge, study their mutual synergy for scaling NPE detectors via preprocessing. The key novelty is three-fold:
  - We propose lightweight VFA and NCA as the preprocessing procedures of NPE detection. They exhibit little overhead but are effective in identifying safe pointer operations.
  - We study the synergistic effects between VFA and NCA for NPE detection. Existing approaches that explore synergistic static analyses are not designed for detecting NPEs.
  - We explore the potential of utilizing simple preprocessing procedures to scale static analysis. As discussed before, this is a method significantly underestimated by existing works.
- We implement our synergistic preprocessing approach as a tool[2] and use it to accelerate two precise static dataflow analyses for detecting NPEs.
- We evaluate our approach using many programs, including those previously used in evaluating KLEE and Pinpoint, showing the effectiveness with a dozen previously unknown NPEs detected and confirmed.

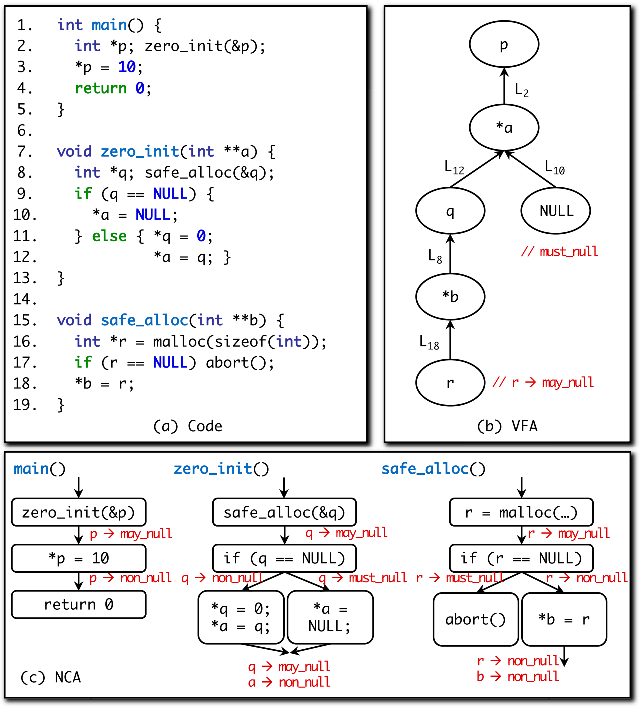
The remainder of the paper is organized as follows. §II provides an overview and §III explains the details of our approach. We discuss the evaluation results in §IV. §V surveys the related work and §VI concludes.

II. Synergistic Effects in a Nutshell

In this section, we use a motivating example to (1) introduce two preprocessing procedures for NPE detection, (2) illustrate the weakness of independently running them, and (3) show how our approach, i.e., BONA, activates their synergistic effects. We refer to the two preprocessing procedures as value-flow analysis (VFA) and null-check analysis (NCA).

[1]Given two consecutive statements dereferencing the same pointer, e.g., `*v = x; *v = y`, the pointer v is deemed nonnull because, if it is null, the code has been crashed at the first statement and cannot reach the second.

[2]https://github.com/BONA-Analyzer/bona

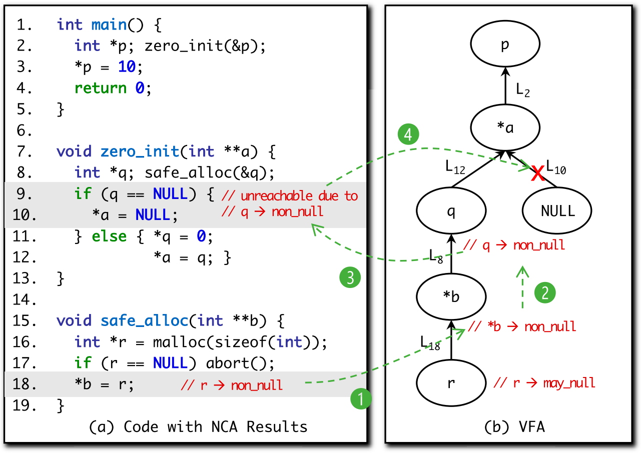


```
1.  int main() {
2.    int *p; zero_init(&p);
3.    *p = 10;
4.    return 0;
5.  }
6.
7.  void zero_init(int **a) {
8.    int *q; safe_alloc(&q);
9.    if (q == NULL) {
10.     *a = NULL;
11.   } else { *q = 0;
12.            *a = q; }
13. }
14.
15. void safe_alloc(int **b) {
16.   int *r = malloc(sizeof(int));
17.   if (r == NULL) abort();
18.   *b = r;
19. }
```

Fig. 2. A motivating example.

Fig. 2(a) shows the code snippet of the motivating example, which is simplified from real programs. The code consists of three functions, `main`, `zero_init`, and `safe_alloc`. The function `safe_alloc` is a wrapper of the C library function `malloc`. It tries to allocate a memory space and returns the pointer of the memory space. If the memory allocation fails, `malloc` returns null, and the program aborts at Line 17. The function `zero_init` invokes the function `safe_alloc` at Line 8 to create a memory space pointed-to by the pointer `q`. It then zero-initializes the memory space at Line 11. The function `main` invokes the function `zero_init` to create an integer (Line 2), assigns it a constant ten (Line 3), and then returns (Line 4). Since the code aborts at Line 17 when `malloc` returns a null pointer, the pointer `p` at Line 3 cannot be null. Hence, NPE cannot happen at Line 3.

VFA. The global value-flow analysis captures the def-use relations in the code. Like many previous works [20], [21], VFA is built on top of a flow- and context-insensitive pointer analysis to capture the def-use relations hidden behind pointer operations and across the function boundaries. As a preprocessing procedure, although VFA is inter-procedure, it remains cheap because it only relies on a "flow- and context-insensitive" pointer analysis. Fig. 2(b) shows the value-flow graph (VFG) that captures the def-use relations. A VFG node represents a variable definition and a VFG edge represents the value propagation between variables. The edge is labeled by L_i if the propagation happens through the code at Line i. In the VFG, the edges labeled by L_8 and L_2 capture the def-use relations hidden by pointer operations and discovered by the pointer analysis. The other edges are direct def-use relations. To check if NPE can



```
1.  int main() {
2.    int *p; zero_init(&p);
3.    *p = 10;
4.    return 0;
5.  }
6.
7.  void zero_init(int **a) {
8.    int *q; safe_alloc(&q);
9.    if (q == NULL) { // unreachable due to
10.     *a = NULL;     // q → non_null
11.   } else { *q = 0;
12.            *a = q; }
13. }
14.
15. void safe_alloc(int **b) {
16.   int *r = malloc(sizeof(int));
17.   if (r == NULL) abort();
18.   *b = r;      // r → non_null
19. }
```

Fig. 3. Steps of achieving the synergistic effects.

happen at Line 3, VFA checks if the pointer `p` at Line 3 may be null. Since the node `p` in the VFG is reachable from possible null pointers, i.e., the node `NULL` and the node `r`, VFA has to conclude that the pointer `p` could be null and reports an NPE at Line 3. As discussed before, this is a false warning.

NCA. Unlike VFA that is global but keeps cheap by resolving flow-insensitive pointer relations, NCA stays cheap by a local but flow-sensitive analysis that does not resolve pointer relations (This is because a global flow-sensitive pointer analysis is often expensive). NCA discovers non-null pointers by two simple rules. In the example, NCA analyzes the control-flow graph of each function individually as illustrated in Fig. 2(c). In the beginning, we assume all pointers could be null. First, whenever a conditional branch with a comparison to null is met, null or non-null facts are derived for the corresponding branch selections. For instance, the pointer `q` cannot be null if we take the false branch of the if-statement at Line 9 but must be null if we take the true branch. Second, when a dereferencing instruction is met, a non-null fact is generated. For instance, the pointer q is deemed not null at Line 12 because it has been dereferenced at Line 11. Fig. 2(c) shows the results of NCA on the example program. From the results, NCA can only conservatively conclude that the pointer `p` in the main function could be null. Therefore, such an independent NCA also mistakenly reports an NPE at Line 3.

BONA. As illustrated above, using NCA or VFA alone fails to filter out the NPE candidate at Line 3. If a preprocessing procedure cannot remove a majority of NPE candidates in a program, the whole analysis will not be scalable, because we then have to rely on an expensive verification procedure to check if such NPE candidates may happen at runtime. Fortunately, by combining NCA and VFA, the synergistic effects enable us to remove the NPE candidate easily and let us avoid an expensive verification procedure. The steps of achieving synergistic effects are shown in Fig. 3.

(1) Boosting VFA by NCA. We first perform NCA for all functions. Since NCA has not obtained any facts from VFA yet, the same facts will be generated as those in Fig. 2(c), where we

get the information: the pointer `r` cannot be a null pointer at the statement `*b = r` (Line 18). VFA then updates the null facts of the VFG nodes as shown by Step ❶ in Fig. 3. That is, the pointer `*b` in the VFG cannot be null because the edge from `r` to `*b` is labeled by L_18, i.e., `*b` receives the value from `r` at Line 18. Since the pointer `q` is only reachable from the non-null pointer `*b`, we can conclude that the pointer `q` cannot be null, as shown by Step ❷ in Fig. 3.

(2) Boosting NCA by VFA. We have concluded based on the boosted VFA that the pointer `q` in the function `zero_init` is not null. Hence, we rerun NCA on the function `zero_init` with this new information. As shown by Step❸ in Fig. 3, since the pointer `q` is not null, NCA does not enter the true branch at Lines 9 and 10. This further means that, in the VFG, the edge labeled by L_{10} can be removed as illustrated by Step ❹. As a result, in the VFG, the pointer `p` is no longer reachable from the constant null pointer.

(3) BONA's Result. After the synergistic analysis above, as shown in Fig. 3(b), the pointer `p` is not reachable from any null pointers. Hence, we can conclude that the pointer `p` is not null and the NPE candidate at Line 3 can be pruned. No further costly analysis is needed to verify the safety of the pointer-dereferencing operation. In practice, we expect to use the synergistic effects between NCA and VFA to boost existing static NPE analysis as the synergistic effects can easily discover a lot more non-null pointers compared to using NCA and VFA independently.

Summary. As illustrated above, our approach has the following merits for NPE detection.

- **Easy to Deploy**: BONA utilizes simple preprocessing procedures to remove NPE candidates without any complex interaction with the follow-up verification procedure. This makes it easy to integrate BONA with any precise but costly NPE detector.
- **Effective for NPE Detection**: The mutual synergy of NCA and VFA, as illustrated above, allows us to remove a majority of NPE candidates before a costly verification procedure and, therefore, improves the scalability of the whole NPE detector.

III. Synergistic Preprocessing

In this section, we first detail the value-flow analysis (VFA, §III-A) and the null-check analysis (NCA, §III-B). Then, we discuss how they achieve mutual synergy (§III-C) and speed up NPE detection (§III-D).

Abstract Language. To ease the detailed explanation of our approach, with no loss of generality, we model the target programs using the following C-like call-by-value language.

In the small language, a program is composed of multiple functions and each function consists of multiple simple or compound statements, including assignments, loads, stores, null comparisons, calls, returns, sequencing, branching, and looping. The semantics of these program statements are standard and, thus, are omitted.

$$
\begin{array}{rcll}
\textit{Program } P & := & F^{+} & \\
\textit{Function } F & := & f(v_1, v_2, \dots)\{\ S;\ \} & \\
\textit{Statement } S & := & v_1 \leftarrow v_2 & \textbf{::assignment} \\
& & \mid v_1 \leftarrow *v_2 \mid *v_1 \leftarrow v_2 & \textbf{::load/store} \\
& & \mid v_1 \leftarrow (v_2 = \mathit{null}) & \textbf{::cmp-to-null} \\
& & \mid r \leftarrow f(v_1, v_2, \dots) & \textbf{::call} \\
& & \mid \textbf{return } v & \textbf{::return} \\
& & \mid S_1; S_2 & \textbf{::sequencing} \\
& & \mid \textbf{if } (v)\ \{\ S_1;\ \}\ \textbf{else}\ \{\ S_2;\ \} & \textbf{::branching} \\
& & \mid \textbf{while } (v)\ \{\ S;\ \} & \textbf{::looping}
\end{array}
$$

Fig. 4. A small C-like language to illustrate our approach.

Abstract Domain. During our static analysis, either VFA or NCA, each program variable v is assigned an abstract value $\hat{v} \in \{\text{MAY-NULL}, \text{MUST-NULL}, \text{MUST-NONNULL}\}$, meaning the dataflow facts that v may be a null pointer, must be a null pointer, or must be a non-null pointer. We define the following meet operation $\sqcap$ between these abstract values,

$$\hat{v}_1 \sqcap \hat{v}_2 = \begin{cases} \hat{v}_1 & \hat{v}_1 = \hat{v}_2 \\ \text{MAY-NULL} & \hat{v}_1 \neq \hat{v}_2 \end{cases}$$

so that the abstract values form a lattice of finite height and guarantee the convergence of static analysis [28].

A. Lightweight Value-Flow Analysis

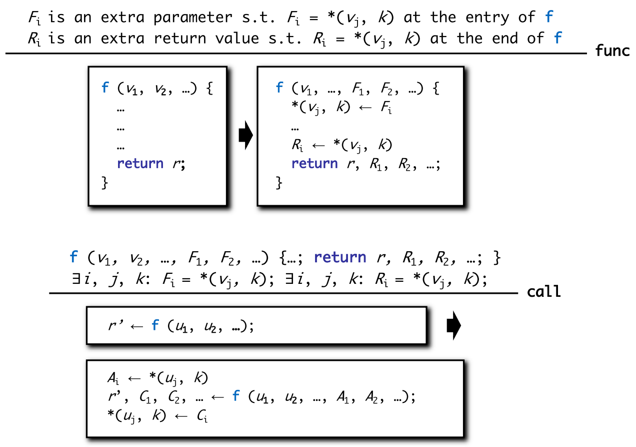
Our approach first builds intra-procedural value-flow graphs (VFGs) for each function. The VFGs are then connected to each other to form a whole-program VFG, on which a whole-program value-flow analysis (VFA) is performed. In general, VFA consists of two tasks. First, VFA builds a VFG that captures the def-use relations among all program variables, including the direct def-use chain as well as those hidden behind pointer operations and across function boundaries. Second, VFA propagates dataflow facts along paths in the graph. Our VFA is cheap because of the following two designs, which are briefly listed below and detailed later.

- Unlike existing works that build complex intermediate representations like the memory SSA [29], we directly build VFG based on a flow- and context-insensitive pointer analysis.
- We leverage a fast reachability indexing technique to speed up reachability queries on the control flow graph, thereby achieving partial flow sensitivity.

VFG. Before diving into the details of VFA, we formally define the value-flow graph that VFA builds, as follows.

Definition 1 (Value-Flow Graph (VFG)): VFG is a directed graph $(\mathbb{V}, \mathbb{E})$ where each vertex $v \in \mathbb{V}$ is a program variable and an edge $(v_1, S, v_2) \in \mathbb{E}$ is a def-use relation, meaning that the value of the variable v_1 flows to the variable v_2 via the program statement S.

To recognize def-use relations hidden behind pointer operations, we first perform a sound, unification-based, flow-insensitive, and context-insensitive pointer analysis to resolve pointer relations in the code [30]. The analysis is of almost linear complexity and, thus, is very cheap. With the pointer analysis results, a standard Mod-Ref analysis is performed to recognize the side effects of every function. Here, side effects have



```
Fi is an extra parameter s.t. Fi = *(vj, k) at the entry of f
Ri is an extra return value s.t. Ri = *(vj, k) at the end of f
-------------------------------------------------------------- func
f (v1, v2, ...) {          f (v1, ..., F1, F2, ...) {
  ...                        *(vj, k) ← Fi
  ...                        ...
  ...                        Ri ← *(vj, k)
  return r;                  return r, R1, R2, ...;
}                          }

f (v1, v2, ..., F1, F2, ...) {...; return r, R1, R2, ...; }
∃i, j, k: Fi = *(vj, k); ∃i, j, k: Ri = *(vj, k);
------------------------------------------------------ call
r' ← f (u1, u2, ...);

Ai ← *(uj, k)
r', C1, C2, ... ← f (u1, u2, ..., A1, A2, ...);
*(uj, k) ← Ci
```

Fig. 5. Transformation rules for function and call statement.

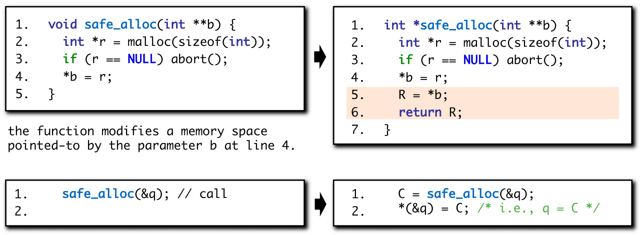
Algorithm 1: VFG

```
1  procedure VFG()
       // Building intra-procedural value flows
2      foreach assignment: $v_1 \leftarrow v_2$ do
3          add edge $(v_2, v_1 \leftarrow v_2, v_1)$ to VFG;
4      foreach load-store pair: $v_1 \leftarrow *v_2, *v_3 \leftarrow v_4$ in a function do
5          if reachable($*v_3 \leftarrow v_4$, $v_1 \leftarrow *v_2$) $\wedge$ alias($v_2$, $v_3$) then
6              let $o$ be a fresh VFG vertex;
7              add edge $(v_4, *v_3 \leftarrow v_4, o)$ into VFG;
8              add edge $(o, v_1 \leftarrow *v_2, v_1)$ into VFG;
       // Building inter-procedure value flows
9      foreach call: $r, C_1, \cdots \leftarrow f(u_1, \cdots, A_1, \cdots)$; do
10         assume the callee $f$ has parameters: $v_1, \cdots, F_1, \cdots$;
11         assume the callee $f$ has returns: $r', R_1, \cdots$;
12         add edges $(u_i, S, v_i)$, $(A_i, S, F_i)$ to VFG where $S$ is the
             call statement;
13         add edges $(r', S, r)$, $(R_i, S, C_i)$ to VFG where $S$ is the
             return statement;
14 procedure alias($v_1$, $v_2$)
       // It is provided by pointer analysis [? ].
          It checks if two inputs are aliases in
          constant time.
15 procedure reachable($S_1$, $S_2$)
       // It is provided by a reachabiliy indexing
          technique [? ]. It checks if the
          statement $S_1$ can reach $S_2$ in a control
          flow graph in almost constant time.
```

a broader meaning, including both referencing and modifying non-local memory locations in a function. With the side-effects, we can transform each function to a pure function by adding extra function parameters and returns, via the transformation rules in Fig. 5.

Specifically, an extra parameter is a variable that stands for a non-local memory location referenced through a pointer expression $*(v, k)$, where v is a formal parameter and we use $*(v, k)$ as a shorthand of dereferencing a pointer k times. Similarly, an extra return value stands for a non-local memory location that is modified in a function. Formally, the transformation rules are shown in Fig. 5. The rule **func** inserts extra parameters F_i and extra return values R_i into a function and establish the relationship between the extra values with the original formal parameters. The rule **call** transforms corresponding function calls after callee functions are transformed.

Example 1: The figure below demonstrates, in practice, how we transform the function `safe_alloc` written in C. The function modifies the memory pointed to by the formal



```
1.  void safe_alloc(int **b) {
2.    int *r = malloc(sizeof(int));
3.    if (r == NULL) abort();
4.    *b = r;
5.  }
```

the function modifies a memory space
pointed-to by the parameter b at line 4.

```
1.  int *safe_alloc(int **b) {
2.    int *r = malloc(sizeof(int));
3.    if (r == NULL) abort();
4.    *b = r;
5.    R = *b;
6.    return R;
7.  }
```

```
1.    safe_alloc(&q); // call
2.
```

```
1.    C = safe_alloc(&q);
2.    *(&q) = C; /* i.e., q = C */
```

parameter `b`. Thus, by the rule **func**, we add an extra return value `R`. Accordingly, by the rule **call**, we also transform the call statement that invokes this function by adding a receiver `C` to receive the extra return value. A store instruction is also inserted to establish the relationship between the pointer `&q` and the receiver `C`. □

After the transformation, all implicit side effects in a function are made explicit via the load and store statements inserted by the transformation rules. As a result, we can easily build VFG using Algorithm 1, which consists of two parts. The first part is from Line 2 to Line 8, which builds intra-procedural value flows. That is, for each assignment $v_1 \leftarrow v_2$, Lines 2–3 build a direct value flow from v_2 to v_1.

In addition to direct value flows, Lines 4–8 build indirect value flows between load and store statements. To this end, given a pair of load and store statements, Line 5 checks if the store statement can reach the load statement and, meanwhile, checks if the load statement loads a value from a memory space that is possibly the same as the memory accessed by the store statement. If both conditions are satisfied, we establish value flows between them. The reachability check is done by a reachability indexing technique [31]. The reachability indexing technique can answer reachability queries in almost constant time after a one-time graph traversal of almost linear complexity, without building an expensive transitive-closure of reachability. The memory-alias check is done by calling the function *alias*(v_1, v_2) provided by the pointer analysis [30]. Since both the reachability check and the alias check are of almost constant complexity and the load and store pairs are restricted in the same function, this procedure is cheap.

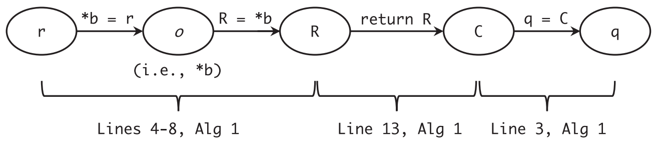
In the procedure above, the load-store pair loop at line 4 may introduce a potentially quadratic blow-up, i.e., $O(n^2)$ complexity where n stands for the number of load and store instructions, if a function has a non-trivial size. On the one hand, most functions in practice are not large. Thus, n is small and does not cause performance issues. On the other hand, we can keep n small via some efficient data structures. For instance, via a linear scan of all instructions, we can put load and store instructions into different small groups. In each group, load and store instructions refer to pointers that are aliases. As such, we only need to enumerate load-store pairs in the small groups.

Algorithm 2: VFA

```
1  $\mathbb{M}(v) \leftarrow$ MAY-NULL for all vertices in VFG $(\mathbb{V}, \mathbb{E})$;
2  $\mathbb{M}(v) \leftarrow$ MUST-NULL for all $v$ that denotes a constant null;
3  $\mathbb{M}(v) \leftarrow$ MUST-NONNULL for all $v$ that cannot be null;
4
5  procedure VFA($\mathbb{E}_{\text{MUST-NULL}}$, $\mathbb{E}_{\text{MUST-NONNULL}}$)
6      $\mathbb{W} \leftarrow \{v : (u, S, v) \in \mathbb{E}_{\text{MUST-NULL}} \cup \mathbb{E}_{\text{MUST-NONNULL}}\}$;
7      while $\mathbb{W} \neq \emptyset$ do
8          $v \leftarrow \mathbb{W}.pop()$;
9          if $\forall (u, S, v) \in \mathbb{E}$ s.t. $(u, S, v) \in \mathbb{E}_{\textit{MUST-NULL}}$ then
10             $\mathbb{M}(v) \leftarrow$ MUST-NULL;
11             $\mathbb{E}_{\text{MUST-NULL}} \leftarrow \mathbb{E}_{\text{MUST-NULL}} \cup \{(v, S', w) \in \mathbb{E}\}$;
12             $\mathbb{W} \leftarrow \mathbb{W} \cup \{w : (v, S', w) \in \mathbb{E}\}$;
13         else if $\forall (u, S, v) \in \mathbb{E}$ s.t. $(u, S, v) \in \mathbb{E}_{\textit{MUST-NONNULL}}$ then
14             $\mathbb{M}(v) \leftarrow$ MUST-NONNULL;
15             $\mathbb{E}_{\text{MUST-NONNULL}} \leftarrow \mathbb{E}_{\text{MUST-NONNULL}} \cup \{(v, S', w) \in \mathbb{E}\}$;
16             $\mathbb{W} \leftarrow \mathbb{W} \cup \{w : (v, S', w) \in \mathbb{E}\}$;
17         else
18             $\mathbb{M}(v) \leftarrow$ MAY-NULL;
```

The second part, i.e., Lines 9–13, builds inter-procedural value-flows for each function call, by adding value flows from the actuals to the formals, and value flows from the return values to their receivers. This part is also lightweight as its complexity is linear in the number of function calls.

Example 2: (Continue.) The figure below shows the VFG snippet built for the transformed code in Example 1. First, we build the indirect value-flow edges from `r` to `R` as per Lines 4–8 in Algorithm 1. We build this indirect value flows because the store and load statements, i.e., `*b = r` and `R = *b`, access the same memory space and the load statement is reachable from the store. The other two edges, which are from `R` to `C` and from `C` to `q`, are direct value flows, which are built as per Line 13 and Line 3 in Algorithm 1, respectively. Note that the value flow path from `r` to `q` is a bit different from that in Fig. 2, where the node `R` and the node `C` are omitted for easing the explanation. □



VFA. VFA determines if a pointer, represented by a VFG vertex, is or is not a null pointer. To this end, we leverage Algorithm 2, a worklist algorithm, to propagate null pointer information on VFG. In detail, we use a map $\mathbb{M}$ to keep the null fact for each pointer. By default, as shown by Line 1 to Line 3 in Algorithm 2, all pointers are conservatively regarded to be possibly null except for those that are constant null pointers or cannot be null pointers, e.g., global addresses and return values of an external function that returns only non-null pointers.

The inputs of VFA are two edge sets, $\mathbb{E}_{\text{MUST-NULL}}$ and $\mathbb{E}_{\text{MUST-NONNULL}}$, representing the edges that do and do not propagate null pointers, respectively (Line 5). When VFA is an independent analysis, the edges are the outgoing edges of the vertices that are definitely null and not null, respectively (see Line 2 and Line 3 in Algorithm 2). Note that the two sets are not empty. For instance, a VFG always contains vertices standing for constant null pointers. Thus, $\mathbb{E}_{\text{MUST-NULL}}$ contains edges outgoing from the vertices standing for constant null. A VFG also may contain vertices standing for pointers that point to global variables, e.g., the pointer `p` in `p = &g` where `g` is a global variable. Such global variable pointers cannot be null, and $\mathbb{E}_{\text{MUST-NONNULL}}$ contains edges from vertices representing them. When we link VFA and NCA as discussed in §III-C, VFA receives the two sets from NCA, which will be detailed later.

Line 6 initializes a worklist that contains all pointers that are successors of a pointer in $\mathbb{E}_{\text{MUST-NULL}}$ and $\mathbb{E}_{\text{MUST-NONNULL}}$. Intuitively, we will check in the follow-up loop if a pointer in the worklist is or is not a null pointer based on the following rule. That is, if all predecessors of a pointer v must (resp. must not) be null, the pointer v must (resp. must not) be null (Line 10 and Line 14). Otherwise, a pointer is regarded as possibly null (Line 17).

Lemma 1: VFA, i.e., Algorithm 2, is sound: if a pointer v may be a null pointer at runtime, we have $\mathbb{M}(v) =$ MAY-NULL after Algorithm 2.

Proof: (Sketch) First, VFG is built on a pointer analysis to figure out the indirect def-use relations at stores, loads, and calls. Since the pointer analysis is sound, i.e., does not miss any possible pointer aliases, the VFG is sound, i.e., contains all possible value flows that may happen at runtime. Second, Algorithm 2 ensures that all pointers reachable from a possibly null pointer in VFG are set to be possibly null. Thus, Algorithm 2 is also sound, i.e., does not miss any possibly null pointers. □

B. Lightweight Null Check Analysis

Our null-check analysis (NCA) aims to check if a pointer `p` dereferenced at a program statement is nullable. For instance, given two consecutive statements dereferencing the same pointer, e.g., `*p = x; *p = y`, the pointer `p` is deemed nonnull at the second statement because, if it is null, the code has crashed at the first statement and cannot reach the second. Since NCA cares about the order of program statements, we realize it as a flow-sensitive intra-procedure dataflow analysis.

A dataflow analysis is often defined by a set of transfer and merging functions over dataflow facts. A transfer function specifies how we compute dataflow facts when visiting a program statement. A merging function defines how we compute dataflow facts at the join point of multiple program paths. Fig. 6 lists the transfer and merging functions as a series of inference rules. In each rule, the part above the horizontal line is a set of assumptions and, under these assumptions, the bottom part describes dataflow facts before and after a statement S (defined in Fig. 4) in the form of $\mathbb{R}, \mathbb{N} \vdash S : \mathbb{R}', \mathbb{N}'$. Here, $\mathbb{R}$ is a set of reachable statements and $\mathbb{N}$ maps each pointer v to a dataflow fact $\hat{v}$ including MAY-NULL, MUST-NULL, and MUST-NONNULL. In the inference rules, we use $\mathbb{N}[v \mapsto \hat{v}]$ to mean we update the dataflow fact in the map.

$$\frac{\mathbb{R} = \emptyset}{\vdash \_ : \mathbb{R}, \mathbb{N}[\forall v : v \mapsto \text{MAY-NULL}]}\ \textbf{init} \qquad \frac{}{\mathbb{R}, \mathbb{N} \vdash v_1 \leftarrow v_2 : \mathbb{R} \cup \{v_1 \leftarrow v_2\}, \mathbb{N}[v_1 \mapsto \mathbb{N}[v_2]]}\ \textbf{assign}$$

$$\frac{}{\mathbb{R}, \mathbb{N} \vdash v_1 \leftarrow *v_2 : \mathbb{R} \cup \{v_1 \leftarrow *v_2\}, \mathbb{N}[v_2 \mapsto \text{MUST-NONNULL}]}\ \textbf{load} \qquad \frac{}{\mathbb{R}, \mathbb{N} \vdash *v_1 \leftarrow v_2 : \mathbb{R} \cup \{*v_1 \leftarrow v_2\}, \mathbb{N}[v_1 \mapsto \text{MUST-NONNULL}]}\ \textbf{store}$$

$$\frac{S \text{ belongs to other types of simple statements}}{\mathbb{R}, \mathbb{N} \vdash S : \mathbb{R} \cup \{S\}, \mathbb{N}}\ \textbf{others} \qquad \frac{\mathbb{R}_1, \mathbb{N}_1 \vdash S_1 : \mathbb{R}_2, \mathbb{N}_2 \quad \mathbb{R}_2, \mathbb{N}_2 \vdash S_2 : \mathbb{R}_3, \mathbb{N}_3}{\mathbb{R}_1, \mathbb{N}_1 \vdash S_1; S_2 : \mathbb{R}_3, \mathbb{N}_3}\ \textbf{sequencing}$$

$$\frac{\begin{array}{c}\text{The conditional variable } v_1 \text{ is the Boolean result of checking if pointer } v_2 \text{ equals null} \\ \mathbb{R}, \mathbb{N}_1 \vdash S_1 : \mathbb{R}'_1, \mathbb{N}'_1 \quad \mathbb{R}, \mathbb{N}_2 \vdash S_2 : \mathbb{R}'_2, \mathbb{N}'_2 \ \textbf{ where } \ \mathbb{N}_1 = \mathbb{N}[v_2 \mapsto \text{MUST-NULL}], \mathbb{N}_2 = \mathbb{N}[v_2 \mapsto \text{MUST-NONNULL}]\end{array}}{\mathbb{R}, \mathbb{N} \vdash \textbf{if } (v_1)\ \{\ S_1;\ \}\ \textbf{else}\ \{\ S_2;\ \} : \begin{array}{ll} \mathbb{R}'_1, \mathbb{N}'_1 & \textbf{if } \mathbb{N}(v_2) = \text{MUST-NULL} \\ \mathbb{R}'_2, \mathbb{N}'_2 & \textbf{if } \mathbb{N}(v_2) = \text{MUST-NONNULL} \\ \mathbb{R}'_1 \cup \mathbb{R}'_2, \mathbb{N}'_1 \cap \mathbb{N}'_2 & \textit{Otherwise} \end{array}}\ \textbf{branching-null-cmp}$$

$$\frac{\begin{array}{c}\text{The conditional variable } v_1 \text{ is not a Boolean result of checking if pointer } v_2 \text{ equals null} \\ \mathbb{R}, \mathbb{N} \vdash S_1 : \mathbb{R}_1, \mathbb{N}_1 \quad \mathbb{R}, \mathbb{N} \vdash S_2 : \mathbb{R}_2, \mathbb{N}_2\end{array}}{\mathbb{R}, \mathbb{N} \vdash \textbf{if } (v_1)\ \{\ S_1;\ \}\ \textbf{else}\ \{\ S_2;\ \} : \mathbb{R}_1 \cup \mathbb{R}_2, \mathbb{N}_1 \cap \mathbb{N}_2}\ \textbf{branching-others}$$

$$\frac{\begin{array}{c}\textit{let: } \mathbb{R}_0, \mathbb{N}_0 \leftarrow \mathbb{R}, \mathbb{N} \quad \textit{repeat: } \mathbb{R}_{i \geq 0}, \mathbb{N}_i \vdash S : \mathbb{R}'_i, \mathbb{N}'_i \quad \mathbb{R}_{i+1}, \mathbb{N}_{i+1} \leftarrow \mathbb{R} \cup \mathbb{R}'_i, \mathbb{N} \cap \mathbb{N}'_i \\ \textit{until: } i = \lambda \textit{ such that we reach the fixed point or, if } v_1 \leftarrow (v_2 = \textit{null}), \mathbb{N}_\lambda(v_2) = \text{MUST-NONNULL}\end{array}}{\mathbb{R}, \mathbb{N} \vdash \textbf{while } (v_1)\ \{\ S;\ \} : \begin{array}{ll} \mathbb{R}_\lambda, \mathbb{N}_\lambda[v_2 \mapsto \text{MUST-NONNULL}] & \textbf{if } v_1 \leftarrow (v_2 = \textit{null}) \\ \mathbb{R}_\lambda, \mathbb{N}_\lambda & \textit{Otherwise} \end{array}}\ \textbf{looping}$$

Fig. 6. Inference rules for NCA.

As shown by the rule **init**, where _ means the entry of a function, at the beginning of analyzing a function, the set of reachable program statements is empty, and the dataflow facts of all variables are set to MAY-NULL, meaning that every pointer may be null. All other rules deal with a simple or compound statement in our small language. The rule **assign** means that after an assignment, the dataflow fact of v_1 propagates to v_2. The rules for **load** and **store** mean that after dereferencing a pointer, the pointer is regarded to be not null. For other simple statements, we do not change the dataflow facts as shown by the rule **others**. For each simple statement, we add it to the set $\mathbb{R}$ after visiting it.

The rule **sequencing** is straightforward. It means that we analyze program statements in order, i.e., using the result of S_1 as the precondition of S_2. The rule **branching-null-cmp** describes how we handle if-statements that check if a pointer is null. In this rule, $\mathbb{N}$ contains the dataflow facts before the if-statement, $\mathbb{N}_1$ and $\mathbb{N}_2$ contain the dataflow facts at the beginning of the true and the false branches, respectively. That is, since the conditional variable v_1 in the if-statement represents the Boolean result of checking if the pointer v_2 is a null pointer, we have $\mathbb{N}_1$ equals $\mathbb{N}[v_2 \mapsto \text{MUST-NULL}]$ and $\mathbb{N}_2$ equals $\mathbb{N}[v_2 \mapsto \text{MUST-NONNULL}]$. If the pointer v_2 must (not) be null, we only take the true (false) branch and the resulting dataflow fact is $\mathbb{N}'_1$ ($\mathbb{N}'_2$). In other cases, we compute the intersection of the two branches, i.e., $\mathbb{N}'_1 \cap \mathbb{N}'_2$, as the analysis result. In terms of the set of reachable statements, if only the true (resp. false) branch is reachable, the resulting set is $\mathbb{R}'_1$ (resp. $\mathbb{R}'_2$). If both branches are reachable, the resulting set is $\mathbb{R}'_1 \cup \mathbb{R}'_2$. The rule **branching-others** deals with other if-statements. We consider both branches to be reachable. Thus, the resulting sets are $\mathbb{R}_1 \cup \mathbb{R}_2$ and $\mathbb{N}_1 \cap \mathbb{N}_2$. The rule **looping** deals with the while loop and involves a fixed-point computation. That is, we repeat analyzing the loop body until $\mathbb{R}_i$ and $\mathbb{N}_i$ do not change or until the loop condition does not hold and we have to exit the loop.

After performing NCA over a function, we have a set of reachable statements $\mathbb{R}$ and a map $\mathbb{N}$ at each program point. Thus, we can check if a reachable program statement may dereference a null pointer.

Example 3: See Fig. 2(c) in §II as an example. □

Lemma 2: NCA, as illustrated by the rules in Fig. 6, is sound, i.e., does not miss any possible null pointers.

Proof: (Sketch) It is easy to check that all rules in Fig. 6 are sound because each rule models the exact semantics of a program statement. Meanwhile, the finite number of statements and the finite height of the lattice ensure the termination of applying the rules. As such, we can obtain the fixed-point results, where $\mathbb{N}$ over-approximate the nullity properties. Hence, NCA, which is performed based on these rules, is sound. □

C. Achieving Synergistic Effects

Simply taking the union of the results of VFA and NCA does not realize the potential of both analyses. We found that VFA and NCA can mutually promote the performance of each other. VFA can refine VFG according to the reachable statements from NCA and can get MUST-NULL and MUST-NONNULL facts to propagate on VFG. NCA, on the other hand, can utilize the information passed by VFA across the function boundaries to perform more precise reasoning.

Boosted VFA. VFA benefits from NCA in two aspects. First, VFG can be refined by NCA. Recall that NCA records reachable statements in a set $\mathbb{R}$ and an edge in VFG is labeled by a program statement. Hence, after NCA, VFG edges labeled by statements not in $\mathbb{R}$ can be removed.

Algorithm 3: BONA

```
1  function BONA()
2      $\mathbb{F} \leftarrow$ all functions;
3      while $\mathbb{F} \neq \emptyset$ do
           // NCA
4          foreach function in $\mathbb{F}$ do
5              perform NCA for each function with the rule
                 init-with-vfa, yielding $\mathbb{R}$ and $\mathbb{N}$ at each statement;
6          $\mathbb{F} \leftarrow \emptyset$;
           // VFA
7          remove VFG edges, $(u, S, v)$ if $S \not\subseteq \mathbb{R}$;
8          build $\mathbb{E}_{\text{MUST-NULL}}$ and $\mathbb{E}_{\text{MUST-NONNULL}}$ based on $\mathbb{N}$;
9          VFA($\mathbb{E}_{\text{MUST-NULL}}$, $\mathbb{E}_{\text{MUST-NONNULL}}$), yielding $\mathbb{M}$;
           // Check if we reach the fixed-point
10         if $\exists v : \mathbb{M}[v]$ changes then
11             add $v$'s function into $\mathbb{F}$;
```

Second, as VFA (see Algorithm 2) accepts two edge sets, i.e., $\mathbb{E}_{\text{MUST-NULL}}$ and $\mathbb{E}_{\text{MUST-NONNULL}}$, as the inputs, NCA can provide more such edges for VFA. That is, since NCA maintains a map $\mathbb{N}$ at every program statement to record if a pointer is null or not null at a program statement. For instance, if $\mathbb{N}(v_2) = \text{MUST-NONNULL}$ at an assignment $v_1 \leftarrow v_2$, the edge $(v_2, v_1 \leftarrow v_2, v_1)$ in VFG (see Line 3, Algorithm 1) should be in the set $\mathbb{E}_{\text{MUST-NONNULL}}$.

Boosted NCA. NCA can receive null-related dataflow facts propagated by VFA. Recall that VFA maintains a map $\mathbb{M}$ that records if a pointer may be, must be, or cannot be null. As such, we can revise the rule **init** in NCA (Fig. 6) as below to receive null-related dataflow facts from VFA.

$$\frac{\mathbb{R} = \emptyset}{\vdash \_ : \mathbb{R}, \mathbb{N}[\forall v : v \mapsto \mathbb{M}[v]]} \quad \textbf{init-with-vfa}$$

BONA. Now we have all the pieces for achieving the synergistic effects. The detailed algorithm is described in Algorithm 3, where VFA and NCA promote each other until reaching the fixed point. The algorithm consists of three parts. The first part, Line 4 to Line 6, performs NCA with the rule **init-with-vfa**, which boosts NCA with VFA results. The second part, Line 7 to Line 9, performs the boosted VFA with NCA results. The third part, Line 10 to Line 11, checks if we reach the fixed point, i.e., if the null-related dataflow facts in a function change. If so, we continue the synergistic preprocessing until the fixed point. Otherwise, we reach the fixed point and terminate the loop.

Although the algorithm runs repetitively until the fixed point is reached, the overhead is negligible compared with running NCA and VFA once. This is because, only in the first round, all functions need to be analyzed by NCA, while in later rounds only a fraction of functions need to be re-analyzed, making the time cost much smaller. When evaluating BONA in §IV, we observe that the loop in Algorithm 3 iterates 2.8 times on average, and every iteration other than the first one takes less than 100ms to finish.

Example 4: See Fig. 3 in §II as an example. □

Lemma 3: BONA, i.e., Algorithm 3, is sound, i.e., does not miss any possible null pointers.

Proof: (Sketch) Algorithm 3 is a composition of VFA and NCA. Since both VFA and NCA are sound by Lemma 1 and Lemma 2, Algorithm 3 is also sound, meaning that it does not miss any MAY-NULL facts and does not generate false MUST-NULL or MUST-NONNULL facts. □

Soundy Implementation. As stated in Lemma 3, BONA is sound in theory with respect to the abstract language in Fig. 4. However, in practice, we have to handle common program structures not included in the abstract language, which leads to a "soundy [32]" (i.e., reasonably unsound) implementation of BONA. In other words, BONA shares the same reasonable assumptions and standard approaches to handle challenging program structures with previous bug-finding techniques, e.g., [5], [6], [7]. For example, following the aforementioned bug-finding techniques, we currently have not modeled inline assembly and call statements that invoke non-standard library APIs. The semantics of standard C/C++ APIs such as `memcpy` and `memset` are manually modeled and embedded in our implementation.

D. Speeding Up Precise but Costly NPE Analyzers

Recall that BONA (Algorithm 3) computes a map $\mathbb{N}$ before each statement. The map $\mathbb{N}$ maps each pointer at a statement to the fact whether the pointer could be null. As such, armed with BONA, an NPE detector can be accelerated in two cases. That is, at a pointer-dereferencing statement, e.g., $v_1 \leftarrow *v_2$, if $\mathbb{N}(v_2) = \text{MUST-NONNULL}$, the NPE detector can skip the procedure of checking NPE at this statement because the pointer dereference must be safe. If $\mathbb{N}(v_2) = \text{MUST-NULL}$, the NPE detector can also skip the procedure of checking NPE at this statement and directly report NPE because the pointer dereference must be unsafe.

In practice, it is unlikely for a pointer at a pointer-dereferencing statement, e.g., v_2 at $v_1 \leftarrow *v_2$, to be marked as MUST-NULL. In our experiments, BONA always marks a pointer at a dereferencing statement as MUST-NONNULL or MAY-NULL. This result follows the intuition that a high-quality program should not contain trivial NPEs that can be easily detected. Thus, the key factor in accelerating NPE detectors is the set of MUST-NONNULL pointers discovered by BONA.

It is worth mentioning that, although MUST-NULL facts rarely propagate to pointer-dereferencing statements, we do not remove MUST-NULL from our abstract domain and do not merge MUST-NULL into MAY-NULL. On the one hand, merging MUST-NULL into MAY-NULL only lets us maintain one less set of pointers, which does not reduce the time and space complexity of our analysis. On the other hand, MUST-NULL are generated at many places, such as null initialization statements like `p=NULL`, and the true branch of null checks `if(p==NULL)`. These MUST-NULL facts are crucial for determining the reachability of statements and deriving MUST-NONNULL facts. For example, in the code below, assuming a fact that p must be null propagates to line 1, the dereference of the pointer r at line 2 will always be executed. As such, our algorithm derives a MUST-NONNULL fact at line 3 — the pointer r is not null at line 3. If we downgrade MUST-NULL to MAY-NULL, we will miss this MUST-NONNULL fact.

```
if (p == NULL)
    *r = 10;
*r = 0;
```

In what follows, we discuss how BONA speeds up two common categories of static analyzers — symbolic execution [9] and path-sensitive dataflow analysis [6], [33].

Symbolic Execution. Symbolic execution techniques like KLEE explore program paths and represent memory states using logical constraints [9]. Using KLEE as an example, it maintains logical constraints that precisely model the effect of each program statement on the memory. When targeting a specific type of bug, such as an NPE, symbolic execution checks whether the memory state can imply a potential null value for a dereferenced pointer, which typically involves constraint solving with an SMT solver. The presence of a large number of program paths can result in a pointer being examined multiple times or forming verbose constraints and, thus, introduce significant overhead to the overall analysis. Armed with BONA, a symbolic-execution-based NPE analyzer like KLEE can check with BONA if $\mathbb{N}(v) =$ MUST-NONNULL at a statement dereferencing the pointer v. If so, we can avoid costly operations like SMT solving for examining the pointer v at that statement, thus reducing the analysis overhead.

Path-Sensitive Dataflow Analysis. Path-sensitive dataflow analysis, exemplified by Pinpoint [6], detects value-flow bugs, including NPEs, by validating the path conditions of program paths connecting sources and sinks in specific forms. For the NPE detection, it needs to collect the path constraints of all reachable paths from a null value to a pointer-dereferencing statement and then solve them, which consumes a significant amount of time. By utilizing the null facts obtained from BONA, the path-sensitive dataflow analysis can avoid collecting and solving the path constraints for the pointer v if $\mathbb{N}(v) =$ MUST-NONNULL at a pointer-dereferencing statement. Clearly, this optimization can help reduce the analysis time.

IV. Evaluation

We aim to, as systematically as possible, evaluate the scaling effect of recent expensive static analyzers boosted by our approach. Particularly, we focus on the study of the following research questions. We also provide a case study at the end to show real NPEs we discovered after arming recent static analyzers with our approach. To conduct the evaluation, we implement BONA and the baseline approaches on top of the LLVM compiler framework. Note that since BONA is designed to improve the "efficiency" rather than "precision" of heavy path-sensitive static analyzers, e.g., KLEE and Pinpoint, the research questions studied in our experiments focus on efficiency improvement. While BONA cannot improve the precision of KLEE and Pinpoint (which already have been very precise due to path-sensitivity) by design, we will show that BONA improves the bug detection capability because due to the improvement of efficiency, BONA enables path-sensitive analyzers to explore more program paths in a given time budget.

- **RQ1:** How many non-null facts can we discover via the synergistic effects?
- **RQ2:** How much overhead does BONA add to existing NPE analyzers?
- **RQ3:** How much can BONA speed up state-of-the-art NPE analyzers?

RQ1. As discussed in §III-D, non-null facts are the key factors to accelerate NPE detection. Thus, we compare the number of non-null facts detected by BONA to the number of non-null facts discovered by VFA, NCA, and their simple combination, denoted as VFA+NCA.

Readers may wonder why we do not compare BONA to a cheap pointer analysis (CPA), e.g., a unification-based flow- and context-insensitive pointer analysis that is widely used as a preprocessing procedure [20], [21]. This is because our VFA is built based on such a CPA. Thus, the results of VFA are almost the same as the CPA's results. In other words, comparing BONA to VFA is almost equivalent to comparing BONA to CPA.

To be more specific, assume the set of pointers dereferenced at a statement S is $\mathcal{D}(S)$. The percentage of nonnull facts discovered in a program P is computed as:

$$\frac{|\{(v,S)|S{\in}P{\wedge}v{\in}\mathcal{D}(S){\wedge}\mathbb{N}(v){=}\text{MUST-NONNULL}\}|}{\Sigma_{S\in P}|\mathcal{D}(S)|}\times 100\%.$$

In the formula, the denominator denotes the total number of pointer-dereferencing operations in a program and the numerator means how many dereferencing operations are safe because the pointer is marked as MUST-NONNULL by BONA. Thus, the larger the percentage is, the fewer pointer-dereferencing operations a downstream NPE detector (see §III-D) needs to check.

RQ2 and RQ3. To address the other two research questions, we use BONA to boost two state-of-the-art static analyzers, KLEE [9] and Pinpoint [6] for NPE detection. Specifically, KLEE is a symbolic execution engine and Pinpoint features a path-sensitive dataflow analysis. The vanilla versions of both tools are expensive as they need to invoke SMT solvers at every pointer-dereferencing statement to check if there are feasible program paths with NPEs. As demonstrated in §III-D, BONA can boost these tools as it can serve as a lightweight preprocessing procedures to discover pointers that cannot be null and, thus, improve the analysis efficiency. We compare the versions boosted by BONA, namely KLEE++ and Pinpoint++, to the versions boosted by VFA+NCA, namely KLEE+ and Pinpoint+, as well as the vanilla versions of KLEE and Pinpoint.

While many static analyzers can check NPEs in addition to KLEE and Pinpoint, we use them in our evaluation because of two reasons. First, we aim to scale expensive static analyzers that detect NPE. KLEE and Pinpoint make use of path-sensitive techniques and, thus, are expensive and belong to the target set of static analyzers we aim to scale. Second, our technique features a preprocessing procedure that is orthogonal to all NPE detectors and can work with any NPE detector. Hence, it is impossible and unnecessary to enumerate and conduct

TABLE I
THE BENCHMARK PROGRAMS

ID	Test Suite	Program Name	KLoC
1	COREUTIL	ls	4.0
2		sort	3.4
3		factor	1.9
4		ptx	1.2
5		csplit	0.8
6		expr	0.8
7		du	0.8
8		tac	0.5
9		nl	0.5
10		cksum	0.2
11	SPEC2006	400.perlbench	128
12		403.gcc	385
13		435.gromacs	85
14		436.cactusADM	60
15		445.gobmk	157
16		454.calculix	75
17		456.hmmer	20
18		464.h264ref	36
19		481.wrf	25
20		482.sphinx3	13
21	REAL	webassembly	23
22		darknet	24
23		html5-parser	31
24		tmux	40
25		libssh	44
26		goaccess	48
27		shadowsocks	53
28		swoole	54
29		libuv	62
30		transmission	88

TABLE II
RQ1: THE PERCENTAGE OF DETECTED NON-NULL FACTS

ID	VFA/CPA	NCA	VFA+NCA	BONA	↑
1	51.9	47.9	63.9	69.2	8.3
2	35.5	43.5	51.1	59.5	16.5
3	55.9	66.2	69.4	75.1	8.4
4	22.1	34.1	39.7	49.8	25.6
5	20.0	27.5	31.7	45.5	43.5
6	14.1	24.7	27.8	40.0	44.2
7	20.3	31.9	35.9	47.4	31.8
8	19.5	27.9	31.6	42.8	35.5
9	20.0	27.1	31.8	43.4	36.1
10	8.6	12.6	15.0	17.4	16.1
11	34.6	39.9	50.0	51.5	3.0
12	34.0	46.7	54.5	58.1	6.7
13	29.5	33.3	40.9	50.9	24.5
14	15.3	44.1	48.2	54.1	12.1
15	96.6	98.6	99.1	99.6	0.5
16	33.6	57.1	63.4	66.0	4.1
17	16.7	22.2	24.2	47.0	94.5
18	35.0	35.7	41.7	55.1	32.0
19	27.1	51.3	56.4	60.3	6.9
20	20.6	25.4	28.7	52.4	82.5
21	19.0	54.4	56.7	64.8	14.3
22	43.3	66.9	69.5	79.9	15.0
23	88.6	89.0	89.1	92.3	3.6
24	38.6	63.7	66.1	71.3	7.9
25	80.4	80.9	81.5	84.6	3.8
26	38.0	65.6	70.8	77.3	9.2
27	33.9	44.0	57.0	63.5	11.3
28	77.4	87.2	87.9	92.3	5.0
29	18.2	56.6	60.4	62.2	3.0
30	42.2	73.7	82.4	89.6	8.7
Avg.	36.4	49.3	54.2	62.1	20.5

experiments on all existing NPE detectors. We use KLEE and Pinpoint as they are two of the most prominent static analyzers that can check NPE.

Benchmark Programs. As listed in Table I, in the evaluation, we include benchmark programs from both KLEE [9] and Pinpoint [6]. They are the ten largest programs from the GNU COREUTILS utility suite, ten largest C programs from the standard SPEC2006 test suite,[3] and ten real-world programs. Note that Pinpoint can analyze programs with or without entry functions; KLEE, which is a symbolic executor, requires an entry function. As the technical part shows, BONA does not assume there is an entry function in the programs to analyze and can work well with both Pinpoint and KLEE. For the first research question, we use all benchmark programs to evaluate the effectiveness of BONA. For the other two research questions, when comparing KLEE to KLEE+ and KLEE++, we use the GNU COREUTILS utility suite, which was used to evaluate KLEE [9]. When comparing Pinpoint to Pinpoint+ and Pinpoint++, we use the SPEC2006 test suite and the real-world programs, which were used to evaluate Pinpoint [6].

Environment. All the experiments are run on a Ubuntu-20.04 server equipped with a 12-core 20-thread Intel Core i5 CPU, with 3.60GHz speed and 64GB of RAM.

[3]We note that the original paper of Pinpoint [6] uses SPEC2000, which is too old and not available to us. We replace it with SPEC2006.

A. RQ1: Non-Null Facts Detected

Recall that our goal is to detect as many non-null facts as possible so that an expensive NPE detector can only focus on a small part of dereferences in the code. Hence, the more non-null facts we can detect, the more effective our approach is. Table II demonstrates the percentage of non-null facts detected by the baselines, i.e., VFA or CPA, NCA, and VFA+NCA, and our approach in the benchmark programs. The percentage represents the number of non-nullable dereferences divided by the total number of dereferences in the code. We can observe that BONA detects up to 94.5%, 19.0% on average, more non-null facts than the simple combination of VFA and NCA. Specifically, BONA detects 49%, 59.5%, and 77.8% non-null facts (17.8%, 33.3%, and 8.2% more non-null facts) on average in the three test suites, respectively. Since we detect more non-null facts than the baseline approaches, as demonstrated in the following subsections, the synergistic effects achieved by BONA can speed up existing static analyzers much more than all other baseline approaches.

As shown in Table II, BONA's improvement over VFA+NCA varies across different programs, from 0.5% to 94.5%. Many program features could affect BONA's performance. We discuss a few as follows. First, the in-degree of VFG could be a measurable characteristic that affects VFA and BONA's effectiveness. That is, when a VFG node has more incoming edges, it has more chances to receive a may-null fact which over-writes any

TABLE III
THE NUMBER OF EXECUTED INSTRUCTIONS BY KLEE, KLEE+, AND KLEE++

ID	KLEE	KLEE+	KLEE++	KLEE vs. KLEE+ (%)	KLEE vs. KLEE++ (%)	KLEE+ vs. KLEE++ (%)
1	1,870,754	2,126,278	2,315,826	13.7	23.8	8.9
2	1,961,781	2,151,820	2,305,456	9.7	17.5	7.1
3	254,521	256,747	284,246	0.9	11.7	10.7
4	6,036	7,884	9,753	30.6	61.6	23.7
5	1,719,187	1,894,182	2,124,930	10.2	23.6	12.2
6	21,326	21,702	22,938	1.8	7.6	5.7
7	24,581	24,581	25,916	0.0	5.4	5.4
8	643,431	671,665	825,205	4.4	28.3	22.9
9	238,088	239,833	240,590	0.7	1.1	0.3
10	437,660	460,937	501,928	5.3	14.7	8.9
Avg.	717,737	785,563	865,679	7.7	19.5	10.6

must-null or must-nonnull facts it gets, making the synergistic effect less effective. When VFG nodes have lower in-degree, the chance for must-null or must-nonnull facts to propagate is bigger. Thus, BONA may perform better. Second, the effectiveness of NCA and BONA could depend on the patterns of the program structure. Patterns like `if (p != null)` or `*p = u` can derive MUST-NONNULL facts that dominate all following dereferences of pointer `p`. The more these patterns exist in the program, the more non-null pointers NCA can infer to boost the effectiveness of BONA. Third, the density of the call graph can affect the effectiveness of BONA. Since NCA is intra-procedure, only when there are more inter-procedural value flows can more must-null or must-nonnull facts computed by NCA be passed across the function boundary, thus increasing the chances of the synergistic effect working.

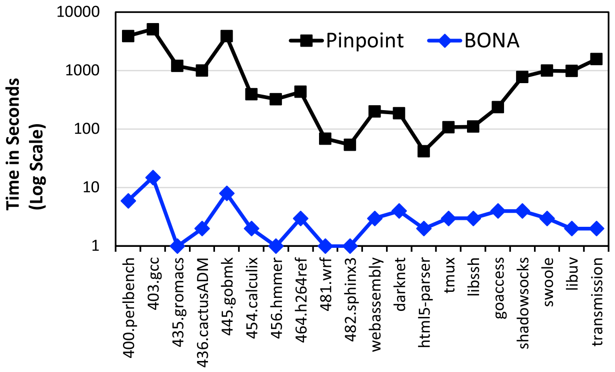


Fig. 7. Time cost of BONA and the original Pinpoint.

B. RQ2: Overhead over NPE Analyzers

In this subsection, we aim to show that, compared to KLEE and Pinpoint, two state-of-the-art path-sensitive static analyzers, the overhead of BONA is negligible. However, as discussed in the next subsection, the performance improvement brought by the small overhead is notable.

1) Using BONA With KLEE: KLEE is a symbolic execution engine that exhaustively explores every program path in the code. When used to detect NPEs, before every pointer-dereferencing instruction, if we can determine the pointer cannot be null, we will skip the NPE detection procedure. Otherwise, the symbolic executor will fork a new execution state, and invoke the SMT solver, to check if an NPE may happen.

When using the original version of KLEE to detect NPE in the ten largest programs from GNU COREUTILS, KLEE cannot complete its analysis in 15 minutes for each program due to the path-explosion problem in symbolic execution. Thus, we try to run BONA to improve the performance of KLEE-based NPE detection, as BONA can help determine if a pointer could be null. For each program, BONA can finish in less than 4 seconds, less than 0.44% (=4/(15*60)) of the time cost of KLEE. In practice, KLEE usually runs for hours or indefinitely when analyzing realistic-sized programs. As reported in [9], for most programs in GUN COREUTILS, KLEE failed to complete within one hour. Thus, in practice, compared to the 0.44% overhead that is computed based on a 15-minute budget, the real overhead added to KLEE by BONA is far less than 0.44%. In conclusion, using BONA with KLEE only adds a negligible overhead to KLEE.

2) Using BONA With Pinpoint: Pinpoint is a sparse data-flow analysis that uses an SMT solver to check if possible NPE paths are feasible or not. Pinpoint works in a bottom-up manner, meaning that it analyzes callees before callers. When analyzing callees, it builds function summaries containing possible NPE paths. When analyzing callers, the function summaries of callees are used to check if NPE may happen at a call site. When detecting NPEs, if we can determine that a pointer cannot be null, we can not only skip the SMT solving procedure that checks path feasibility but also reduce the number of function summaries to build. Thus, a promising preprocessing procedure is very helpful.

Fig. 7 shows the time cost (in seconds) of the original Pinpoint and the preprocessing procedure. As illustrated in the figure and listed in the second column in Table IV, it usually takes Pinpoint 42 to 5,099 seconds to analyze the SPEC and real programs. In contrast, BONA can finish in 15 seconds. BONA's time cost ranges from 0.08% to 4.76% and, on average, 0.32% of Pinpoint's cost. Thus, we can conclude that, using BONA as a preprocessing procedure of Pinpoint only adds a negligible overhead to Pinpoint.

TABLE IV
THE TIME COST (IN SECONDS) OF PINPOINT, PINPOINT+, AND PINPOINT++

ID	Pinpoint	Pinpoint+	Pinpoint++	Pinpoint vs. Pinpoint+	Pinpoint vs. Pinpoint++	Pinpoint+ vs. Pinpoint++
11	3,912	1,901	1,640	2.1x	2.4x	1.2x
12	5,099	3,293	2,792	1.5x	1.8x	1.2x
13	1,200	782	471	1.5x	2.5x	1.7x
14	995	691	424	1.4x	2.3x	1.6x
15	3,881	1,902	589	2.0x	6.6x	3.2x
16	392	164	99	2.4x	4.0x	1.7x
17	326	132	54	2.5x	6.0x	2.4x
18	433	199	66	2.2x	6.6x	3.0x
19	69	36	21	1.9x	3.3x	1.7x
20	54	18	10	3.0x	5.4x	1.8x
21	201	82	54	2.5x	3.7x	1.5x
22	187	63	32	3.0x	5.8x	2.0x
23	42	20	19	2.1x	2.2x	1.1x
24	108	78	30	1.4x	3.6x	2.6x
25	110	45	31	2.4x	3.5x	1.5x
26	237	186	109	1.3x	2.2x	1.7x
27	781	391	156	2.0x	5.0x	2.5x
28	1,001	438	321	2.3x	3.1x	1.4x
29	987	499	309	2.0x	3.2x	1.6x
30	1,567	612	429	2.6x	3.7x	1.4x
Avg.	1,079	577	383	2.1x	3.8x	1.8x

C. RQ3: Performance Improvement for NPE Analyzers

We have shown that the overhead of BONA is small. This subsection aims to demonstrate that, the small overhead brought by BONA can notably improve the performance of KLEE and Pinpoint, two prominent static analyzers that can detect NPE with high precision.

1) Using BONA With KLEE: While there are a few possible measurements, such as the time cost of the symbolic execution and the number of pointers proved to be safe, that can be used to show the performance improvement brought by BONA, we choose to measure the number of instructions executed by KLEE in a 15-minute time budget. On the one hand, we do not use "the time cost of symbolic execution" as the measurement because KLEE cannot finish analyzing the benchmark programs due to path explosion. On the other hand, the number of pointers proved safe is not a proper measurement because symbolic execution typically is not sound and cannot prove the safety of pointers.

Table III shows the number of executed instructions for the largest 10 programs in COREUTILS. We can observe that KLEE++ executes more instructions in the given time budget, which is 15 minutes in our experiment settings. Compared to KLEE, KLEE+ improves efficiency in terms of the number of executed instructions by 0.0% to 30.6% with 9.5% on average (see Column 5 in Table III), and KLEE++ improves the efficiency by 1.1% to 61.6% with 20.6% on average (see Column 6 in Table III). Compared to KLEE+ where VFA and NCA are used independently, KLEE++ takes advantage of the synergistic effects and obtains an efficiency improvement of 0.3% to 23.7%, with 10.2% on average (see Column 7 in Table III).

2) Using BONA With Pinpoint: Similar to the experiments over KLEE, we integrate BONA and the baseline approach VFA+NCA into the Pinpoint static analyzer. Table IV shows the time cost in seconds (including both the preprocessing and Pinpoint's cost) for programs in each test suite. We can observe that Pinpoint++ is much faster than the others, demonstrating the impacts of our approach again. Compared to Pinpoint, Pinpoint+ improves efficiency in terms of time cost by 1.3x to 3.0x with 1.9x on average (see Column 5 in Table IV), and Pinpoint++ improves the efficiency by 1.8 to 6.6x with 2.8x on average (see Column 6 in Table IV). Compared to Pinpoint+ where VFA and NCA are used independently, Pinpoint++ takes advantage of the synergistic effects and obtains an efficiency improvement of 1.1x to 3.2x, with 1.8x on average (see Column 7 in Table IV). Such a speedup is significant enough as it allows us to check more pointer-dereferencing operations in a given time budget and, thus, discover more NPEs.

D. Detected Real Bugs

As listed below, in the experiments, Pinpoint++ discovered in real-world projects 12 NPEs that Pinpoint cannot find. All 12 bugs were previously unknown and have been confirmed by the developers.

Project	webassembly	tmux	shadowsocks	transmission
# NPE	3	3	4	2

While this is not because our approach improves the bug-finding capability if analysis resources were unlimited, BONA improves the analysis efficiency and, thus, allows KLEE and Pinpoint to check more pointer-dereferencing instructions in a given time budget. For instance, by default, Pinpoint sets the timeout of analyzing a function to one minute. An NPE not checked within the time limit will be missed. Fig. 8 illustrates an NPE we detected in the program *shadowsocks*. The code in the figure aims to read a configuration file in JSON and parse the name-value pairs in the JSON file. At Line 11, the code expects the value corresponding to the name "plugin" is a character

```
1.   jconf_t * read_jconf(const char *file) {
2.     …
3.     // parsing a json object, i.e., a name-value pair, read from file
4.     char *name        = obj->u.object.values[i].name;
5.     json_value *value = obj->u.object.values[i].value;
6.     if (…) {
7.       …
8.     } else if  …
9.       …
10.    } else if (strcmp(name, "plugin") == 0) {
11.      conf.plugin = to_string(value);
12.      if (strlen(conf.plugin) == 0) {
13.        ss_free(conf.plugin);
14.        conf.plugin = 0;
15.      }
16.    } else if  …
17.      …
18.    } else if (strcmp(name, "mode") == 0) {
19.      char *mode_str = to_string(value);
20.      if (strcmp(mode_str, "tcp_only") == 0)    // report NPE
21.        conf.mode = TCP_ONLY;
22.      else if (strcmp(mode_str, "tcp_and_udp") == 0)   // not report NPE
23.        conf.mode = TCP_AND_UDP;
24.      else if (strcmp(mode_str, "udp_only") == 0)    // not report NPE
25.        conf.mode = UDP_ONLY;
26.      ssfree(mode_str);
27.    }
28.    …
29.  }
```

Fig. 8. An example of the detected NPEs in the project *shadowsocks*.

string. This may not be true if a malformed configuration file is provided. Hence, Line 11 may return a null pointer to the variable *conf.plugin*, which will be dereferenced at Line 12 via the function *strlen*, leading to a null pointer exception.

A more interesting case is in the code from Line 19 to Line 26. At Line 19, the code expects that the variable *value* is a character string but it is not. Thus, the pointer *mode_str* is assigned a null pointer, which will be dereferenced three times at Line 20, Line 22, and Line 24 by the function *strcmp*. Our NCA strategy will let us skip the pointer-dereferencing instructions at Line 22 and Line 24, because they are dominated by that at Line 20. That is, if we can pass Line 20, the pointer *mode_str* cannot be a null pointer. Thus, we only report the NPE at Line 20. This does not mean we miss the other two reports but makes the bug report concise. Developers only need to fix the NPE at Line 20, e.g., by adding a null check between Line 19 and Line 20, the other two will also be fixed.

V. Related Work

Static bug finding is a classic topic and has been extensively studied in past decades. Among existing techniques, many aim to achieve high precision by inferring and solving path conditions [5], [6], [7], [8], [9], [33], [34], [35], [36]. These techniques are often very expensive and may have to take a few hours to check a software system. To speed up, many optimization techniques have been proposed. First, many static analyses adopt a compositional design as they produce function summaries to avoid repetitive analyses of the same function at different call sites [5], [6], [7], [8], [37], [38], [39]. Second, abstraction-based approaches such as SLAM [40], BLAST [41], and SATABS [42], [43] adopt abstract refinement to improve the scalability. Third, it has been reported sparse dataflow analysis gains scalability by propagating dataflow facts via data dependence, thus being capable of skipping unnecessary control flows [6], [14], [15], [21], [44]. Fourth, it is also effective to speed up static analysis by utilizing the mutual synergy among different bug types [35]. Fifth, we can also design parallel or distributed algorithms to scale static analysis [16], [17], [18]. More recently, Shi et al. [33] proposed an approach to fusing the static analysis and the constraint solver to save the unnecessary cost of computing path conditions and optimizing the constraint solver via program information.

Compared to the aforementioned approaches, it seems to be straightforward to utilize a lightweight and sound analysis as a preprocessing procedure to prune easy cases. Thus, all aforementioned works rarely mention their strategies of preprocessing. However, we find that trivially applying preprocessing analyses could significantly limit their effectiveness in improving the analysis speed. To break out the limits, we propose a more powerful preprocessing strategy, which utilizes the mutual synergy among two sound and lightweight analyses. In addition to KLEE [9] and Pinpoint [6] that have been extensively evaluated in our experiments, we believe our approach is orthogonal to all aforementioned existing works and can be directly employed to accelerate the detection of null exceptions.

In addition to general static bug-finding frameworks discussed above, there are also a few static analyses specially designed for detecting null pointer exceptions. SALSA [12] aims for verification, i.e., to identify all pointer dereferences that can be concluded to be safe; all remaining dereferences may cause null exceptions. SALSA also depends on a scalable imprecise preprocessing analysis, but it does not utilize a synergistic preprocessing procedure like our approach. XYLEM [11] aims for bug detection instead of verification. It implements a backward demand-driven strategy to find the evidence of null exceptions. Such demand-driven approaches can avoid demand-irrelevant computations, thus being more scalable than exhaustive analyses. Similarly, Madhavan and Komondoor [13] proposed a demand-driven analysis for detecting null exceptions. They focus more on how to soundly handle recursive data structures and perform the strong update for pointer analysis.

NullAway [10] and CheckerFramework [45] are the most recent null exception detectors based on type-based static analysis. We do not compare BONA with them in the experiments due to three reasons. First, our implementation is for C/C++, but NullAway and CheckerFramework are implemented for Java. We failed to find their C/C++ version and cannot compare BONA to them directly. Second, except for being designed for different languages, NullAway and CheckerFramework heavily depend on manual annotations on the nullability of program variables. BONA does not rely on any manual efforts. In this sense, BONA is orthogonal to them as BONA may be able to provide automatic annotations: if BONA can annotate nonnull pointers according to its result. Third, as discussed in the introduction, BONA is designed as a preprocessing procedure to scale heavy path-sensitive static analysis. NullAway and CheckerFramework are not heavy path-sensitive analyzers and, thus, out of the scope of our target static analyses.

VI. Conclusion

In this work, we study the potential of utilizing preprocessing procedures to scale precise yet costly NPE analysis. We argue that this is an effective method but significantly underestimated by previous works. Particularly, we propose to utilize the synergistic effects, which allow us to design more powerful preprocessing procedures. The evaluation results demonstrate that we can significantly speed up state-of-the-art static analyzers. We believe that such a synergistic preprocessing procedure has great potential for speeding up the detection of other bug types and other static analyzers, which we leave as our future work.

References

[1] "Stubborn weaknesses in the CWE top 25," Accessed: Sep. 18, 2023. [Online]. Available: https://bit.ly/3Wdmi4E

[2] J. L. Henning, "SPEC cpu2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[3] B. Meyer, "Ending null pointer crashes," *Commun. ACM*, vol. 60, no. 5, pp. 8–9, 2017.

[4] "CVE - search results," Accessed: Feb. 6, 2024. [Online]. Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=null+pointer

[5] Y. Xie and A. Aiken, "Scalable error detection using Boolean satisfiability," in *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL),* New York, NY, USA: ACM, 2005, pp. 351–363.

[6] 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 *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Des. Implementation, (PLDI)*, New York, NY, USA: ACM, 2018, pp. 693–706.

[7] D. Babic and A. J. Hu, "Calysto: Scalable and precise extended static checking," in *Proc. 30th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2008, pp. 211–220.

[8] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv, "Precise and compact modular procedure summaries for heap manipulating programs," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation, (PLDI)*, New York, NY, USA: ACM, 2011, pp. 567–577.

[9] C. Cadar et al., "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Symp. Oper. Syst. Des. Implementation (OSDI)*, USENIX, 2008, pp. 209–224.

[10] S. Banerjee, L. Clapp, and M. Sridharan, "NullAway: Practical type-based null safety for Java," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA: ACM, 2019, pp. 740–750.

[11] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for Java," in *Proc. 31st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2009, pp. 133–143.

[12] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda, "Verifying dereference safety via expanding-scope analysis," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2008, pp. 213–224.

[13] R. Madhavan and R. Komondoor, "Null dereference verification via over-approximated weakest pre-conditions analysis," *ACM Sigplan Notices*, vol. 46, no. 10, pp. 1033–1052, 2011.

[14] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proc. 25th Int. Conf. Compiler Construction (CC),* New York, NY, USA: ACM, 2016, pp. 265–266.

[15] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI),* New York, NY, USA: ACM, 2007, pp. 480–491.

[16] Q. Shi and C. Zhang, "Pipelining bottom-up data flow analysis," in *Proc. 42nd Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2020, pp. 835–847.

[17] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," *ACM SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 5–10, 2010.

[18] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani, "Parallelizing top-down interprocedural analyses," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Des. Implementation(PLDI)*, New York, NY, USA: ACM, 2012, pp. 217–228.

[19] B. Steensgaard, "Points-to analysis in almost linear time," in *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.(POPL),* New York, NY, USA: ACM, 1996, pp. 32–41.

[20] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *Proc. 9th Int. Symp. Code Gener. Optim. (CGO)*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 289–298.

[21] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi, "Design and implementation of sparse global analyses for C-like languages," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, New York, NY, USA: ACM, 2012, pp. 229–238.

[22] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 13, no. 2, pp. 181–210, 1991.

[23] C. Click and K. D. Cooper, "Combining analyses, combining optimizations," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 17, no. 2, pp. 181–196, 1995.

[24] C. Chambers and D. Ungar, "Iterative type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs," in *Proc. 11th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, 1990, pp. 150–164.

[25] S. Lerner, D. Grove, and C. Chambers, "Composing dataflow analyses and transformations," in *Proc. 29th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, New York, NY, USA: ACM, 2002, pp. 270–282.

[26] S.-A.-A. Touati and D. Barthou, "On the decidability of phase ordering problem in optimizing compilation," in *Proc. 3rd Conf. Comput. Frontiers, (CF),* New York, NY, USA: ACM, 2006, pp. 147–156.

[27] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. 2nd Int. Symp. Code Gener. Optim. (CGO)*, Piscataway, NJ, USA: IEEE Press, 2004, pp. 75: 1–75:12.

[28] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA, USA: Addison-Wesley, 2007. [Online]. Available: https://bit.ly/3fkAEKs

[29] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA: ACM, 2012, pp. 254–264.

[30] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su, "Fast algorithms for Dyck-CFL-reachability with applications to alias analysis," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, New York, NY, USA: ACM, 2013, pp. 435–446.

[31] H. Yildirim, V. Chaoji, and M. J. Zaki, "GRAIL: Scalable reachability index for large graphs," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 276–284, 2010.

[32] B. Livshits et al., "In defense of soundiness: A manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[33] Q. Shi, P. Yao, R. Wu, and C. Zhang, "Path-sensitive sparse analysis without path conditions," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation (PLDI)*, New York, NY, USA: ACM, 2021, pp. 930–943.

[34] I. Dillig, T. Dillig, and A. Aiken, "Sound, complete and scalable path-sensitive analysis," in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Des. Implementation (PLDI)*, New York, NY, USA: ACM, 2008, pp. 270–280.

[35] Q. Shi, R. Wu, G. Fan, and C. Zhang, "Conquering the extensional scalability problem for value-flow analysis frameworks," in *Proc. 42nd Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA: ACM, 2020, pp. 812–823.

[36] 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 *Proc. 41st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 72–82.

[37] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *Proc. 10th Eur. Softw. Eng. Conf. (ESEC/FSE),* New York, NY, USA: ACM, 2005, pp. 115–125.

[38] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 388–402, Jun. 2004.

[39] C. Y. Cho, V. D'Silva, and D. Song, "BLITZ: Compositional bounded model checking for real-world programs," in *Proc. 28th Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 136–146.

[40] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with slam," *Commun. ACM*, vol. 54, no. 7, pp. 68–76, 2011.

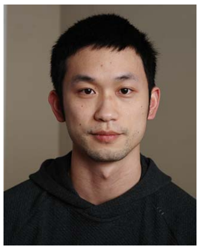
[41] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy abstraction,” in *Proc. 29th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, New York, NY, USA: ACM, 2002, pp. 58–70.

[42] E. Clarke, D. Kroening, and K. Yorav, “Behavioral consistency of C and verilog programs using bounded model checking,” in *Proc. 40th Des. Automat. Conf. (DAC)*, New York, NY, USA: ACM, 2003, pp. 368–371.

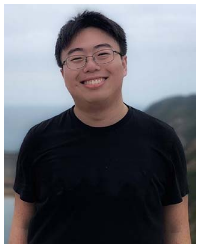
[43] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, “Predicate abstraction of ansi-C programs using SAT,” *Formal Methods Syst. Des.*, vol. 25, no. 2, pp. 105–127, 2004.

[44] Y. Sui, D. Ye, and J. Xue, “Detecting memory leaks statically with full-sparse value-flow analysis,” *IEEE Trans. Softw. Eng.*, vol. 40, no. 2, pp. 107–122, 2014.

[45] M. Kellogg, D. Daskiewicz, L. N. Duc Nguyen, M. Ahmed, and M. D. Ernst, “Pluggable type inference for free,” in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2023, pp. 1542–1554.



Yi Sun received the bachelor’s degree from Nanjing University, China. He is currently working toward the Ph.D. degree with the Computer Science Department, Purdue University. His research focuses on developing techniques using program analysis, particularly static analysis, to detect bugs in traditional software systems, mobile applications (such as Android apps), and blockchain applications. He has contributed to research published at leading software engineering and security conferences such as ICSE, NDSS, and USENIX Security.



Chengpeng Wang received the bachelor’s and master’s degrees from Tsinghua University, in 2016 and 2019, respectively, and the Ph.D. degree from Hong Kong University of Science and Technology, in 2023. He is a Postdoctoral Research Fellow with the Computer Science Department of Purdue University. His research mainly focuses on the use of program analysis, especially static analysis, to improve software reliability and performance. He is also interested in the intersection of machine learning techniques, such as Large Language Models, and symbolic analysis techniques, with the aim of establishing neuro-symbolic program analysis. His contributions to the field have been recognized through publications in esteemed conferences and journals on programming languages, software engineering, and systems. He has been awarded the SIGPLAN Distinguished Paper Award (2022) and the ASPLOS Best Paper Award (2024).

Gang Fan received the Ph.D. degree from Hong Kong University of Science and Technology (HKUST). He is a seasoned Software Analysis Researcher with over 13 years of experience. He currently works at Huawei Research. Previously, he led a team at Ant Group focusing on creating a cutting-edge big code analytic platform. His journey includes co-founding Sourcebrella Inc., a trailblazing firm known for its advanced software analyzer, which was subsequently acquired by Ant Group. He has a strong publishing record, including a paper that received the ACM Distinguished Paper Award. His expertise extends to training CodeFuse, Ant Group’s specialized Large Language Model for coding tasks. This blend of experience and ongoing research contributions continues to fuel his commitment to technological innovation. This work was done when he was with HKUST.

Qingkai Shi (Member, IEEE) received the Ph.D. degree from Hong Kong University of Science and Technology. He was a Postdoctoral Researcher at Purdue University, West Lafayette, IN, USA. He is an Associate Professor with the School of Computer Science, Nanjing University. His research focuses on the use of compiler techniques, especially static program analysis, to rigorously ensure software security. He has published extensively at premium venues of programming languages, cybersecurity, and software engineering. His research received many awards, including an ACM SIGPLAN Distinguished Paper Award, an ACM SIGSOFT Distinguished Paper Award, a Google Research Paper Reward, and the Hong Kong Ph.D. Fellowship. He co-founded Sourcebrella LLC, where his research was commercialized. He then moved to Ant Group as Sourcebrella was acquired.

Xiangyu Zhang (Member, IEEE) is a Professor specializing in AI security, software analysis, and cyber forensics. His work involves developing techniques to detect bugs, including security vulnerabilities, in traditional software systems as well as AI models and systems, and to diagnose runtime failures. He has served as the Principal Investigator (PI) for numerous projects funded by organizations such as DARPA, IARPA, ONR, NSF, AirForce, and industry. Many of the techniques developed by his team have successfully transitioned into practical applications. His research outcome has been published on top venues in the areas of Security, AI, software engineering, and programming languages, and recognized by various distinguished paper awards including the prestigious ACM Distinguished Dissertation Awards. He has mentored over 30 Ph.D. students and postdoctoral, with fifteen securing academic positions in various universities. Many of them have been honored with NSF Career Awards or comparable recognitions.