Precise Compositional Buffer Overflow Detection via Heap Disjointness

Yiyuan Guo
The Hong Kong University of Science and Technology
Hong Kong, China
yguoaz@cse.ust.hk

Peisen Yao
Zhejiang University
Hangzhou, China
pyaoaa@zju.edu.cn

Charles Zhang
The Hong Kong University of Science and Technology
Hong Kong, China
charlesz@cse.ust.hk

ABSTRACT

Static analysis techniques for buffer overflow detection still struggle with being scalable for millions of lines of code, while being precise enough to have an acceptable false positive rate. The checking of buffer overflow necessitates reasoning about the heap reachability and numerical relations, which are mutually dependent. Existing techniques to resolve the dependency cycle either sacrifice precision or efficiency due to their limitations in reasoning about symbolic heap location, i.e., heap location with possibly symbolic numerical offsets. A symbolic heap location potentially aliases a large number of other heap locations, leading to a disjunction of heap states that is particularly challenging to reason precisely.

Acknowledging the inherent difficulties in heap and numerical reasoning, we introduce a disjointness assumption into the analysis by shrinking the program state space so that all the symbolic locations involved in memory accesses are disjoint from each other. The disjointness property permits strong updates to be performed at symbolic heap locations, significantly improving the precision by incorporating numerical information in heap reasoning. Also, it aids in the design of a compositional analysis to boost scalability, where compact and precise function summaries are efficiently generated and reused. We implement the idea in the static buffer overflow detector Cod. When applying it to large, real-world software such as PHP and QEMU, we have uncovered 29 buffer overflow bugs with a false positive rate of 37%, while projects of millions of lines of code can be successfully analyzed within four hours.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**.

KEYWORDS

Static analysis, bug detection, buffer overflow.

ACM Reference Format:
Yiyuan Guo, Peisen Yao, and Charles Zhang. 2024. Precise Compositional Buffer Overflow Detection via Heap Disjointness. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652110

1 INTRODUCTION

Thirty-five years after the Morris worm, buffer overflow remains a largely unsolved problem today, causing severe threats to the correctness and safety of software. As an example, OpenSSL, one of the foundational building blocks for secure computing over the Internet, suffers from buffer overflow bugs constantly: the notorious Heartbleed vulnerability [70] affected two-thirds of Web servers in 2014, and critical buffer overflows capable of "Heartbleed level damage" are still being patched recently in OpenSSL [55].

While dynamic analysis, such as fuzzing, has helped detect a large number of buffer overflows in the wild (e.g., over 10000 overflow-induced crashes were found by OSSFuzz [62]), static analysis for buffer overflow has made very limited progress in the past decade. The most recent effort [48] of this line of work scaled to large programs at the cost of essentially giving up the precise reasoning of pointers. However, the ever-growing size of modern software requires static buffer overflow detection to go through millions of lines of code, doing so in a matter of hours, and detecting bugs hidden behind deep and complex dataflow [3, 24] with a low false positive rate.

1.1 Problem: Mutually Dependent Properties

The static analysis of buffer overflow needs to reason about both the heap reachability (where the accessed pointer points to) and numerical relations (the offset value of memory location w.r.t the bound of the pointed memory object). While each aspect on its own is already challenging, the two properties could be mutually dependent such that imprecision from one side significantly affects the other, leading to false positive reports in buffer overflow detection.

We use the code in Fig. 1 for illustration. A buffer `buf` of size `SZ` is allocated at Line 10 and the statements at Line 13 and 14 accesses `buf` at index 0 and `SZ` respectively. Hence, Line 13 is safe but Line 14 triggers an off-by-one buffer overrun.

Let's consider how should the static analyzer handle the function `set_mem`. The function stores at the heap location `&ar[x]`, which may or may not overwrite the content at `&ar[0]` depending on the numerical value of `x` (Line 7 of Fig. 1). Indeed, reasoning the innately unbounded heap [37] requires numerical information to perform strong updates [66], which is critical for the precision [21]. Meanwhile, analysis of the numerical relations necessitates a model of the heap to resolve memory dependencies [26, 27]. To determine the numerical value for the variables `t1` and `t2` at Line

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652110

12, heap reasoning is required to infer the possible values loaded from `&buf[0]` and `&buf[1]`.

To precisely account for the mutual dependence between the heap and the numerical values, the analysis needs to distinguish among symbolic heap locations, i.e., heap locations with symbolic numerical offsets. Essentially, we need to reason about a disjunction of heap states based on the numerical information, e.g., storing to `&ar[0]` and `&ar[x]` at Line 7 of Fig. 1 results in two different heap states depending on whether `x=0`. Such disjunctive heap reasoning is crucial in our example to avoid false positive: if `&ar[0]` and `&ar[x]` are not distinguished, then it is conservatively inferred that `t1`, `t2` $\in \{0, 1\}$ after the call to `set_mem` at Line 11, leading to a false positive report of buffer underrun at Line 13 (the index `t1-1` may equal -1). However, existing works exhibit a tension between precision and efficiency regarding such disjunctive reasoning about heap states, which we now describe in detail.

Conservative memory model hurts the precision. To break the dependency cycle, most existing approaches [28, 45, 48, 51, 71, 75] either abandon the reasoning of pointers entirely [71, 75] or take a layered approach: an off-the-shelf pointer analysis is utilized to disambiguate indirect memory references before performing the numerical analysis [28, 45, 48, 51]. Their memory models are conservative for being oblivious to the numerical state of the program: the set of symbolic heap locations derived from the same base address is approximated with a summary location even though they may have diverse numerical offsets, essentially joining the disjunction of heap states into a sound over-approximation. This approach is efficient but can lead to significant imprecision, e.g., `&ar[0]` and `&ar[x]` at Line 7 of Fig. 1 are treated as aliases.

Although several static analyzers [6, 53] adopt a product domain where the numerical and heap states are updated together during abstract interpretation [17], they do not utilize the numerical state to materialize [60] the heap abstraction for disjunctive reasoning. Instead, states for symbolic heap locations such as `&ar[0]` and `&ar[x]` are also conservatively joined.

Precise symbolic methods hurts the performance. To precisely analyze `set_mem`, a case split over the heap abstraction is needed to produce the two heap states shown in Fig. 2a based on the value of `x`, which results in an exponential blow-up if more heap operations were performed [22]. Methods based on symbolic model checking implicitly perform the disjunctive reasoning of heap states by encoding the semantics of loading and storing at symbolic heap locations with logical constraints, e.g., the array theory [8, 34, 57] and the index qualifying constraints for the points-to edges [21] (an example is shown in Fig. 2b). The constraints serve as verification conditions, which may eventually lead to a similar disjunctive explosion in the solver. For example, decision procedures for the array theory may need to perform massive case analyses for the numerical indices to instantiate the array axioms [29, 65], causing high-performance overhead. Despite the recent advances in optimizing constraint solving [30, 56, 65, 77], these techniques are still better suited for the formal verification of small- or medium-sized programs instead of detecting bugs in large codebases.

```
1  int* mem_alloc(unsigned sz) {
2    int *buf = malloc(sizeof(int) * sz);
3    return buf;
4  }
5  #define SZ 100
6  void set_mem(int *ar, int x) {
7    ar[0] = x; ar[x] = 0;
8  }

9  void baz(int arg) {
10   int *buf = mem_alloc(SZ);
11   set_mem(buf, 1);
12   int t1 = buf[0], t2 = buf[1];
13   buf[t1-1] = 0;
14   buf[t2+SZ] = 0;
15 }
```

Figure 1: A motivating example.

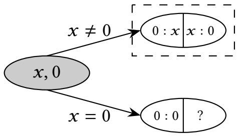
1.2 Our Goal And Solution

In this paper, we make no claims about resolving the innate difficulties of the verification for heap and numerical properties. Inspired by the recent success of under-approximate static analysis in bug catching of real systems [5, 44], our goal is to design a practical static analysis that efficiently finds a subset of buffer overflow bugs with flow-, context-, and path-sensitivity, while capable of distinguishing various symbolic heap locations.

Our key insight is that while precisely reasoning about the memory operations at symbolic heap locations can involve a large disjunction of heap states, the static analysis can focus on a subset of the disjunction where the tracked symbolic heap locations are ***pairwise disjoint***. Indeed, the need for case splits over the heap abstraction stems from the potential aliasing among the heap locations [4], i.e., different aliasing patterns need to be considered when handling memory operations. The disjointness property boosts both the precision and efficiency of our analysis:

- Without needing any case analysis, strong updates are enabled to symbolic heap locations to improve the precision of heap reasoning, which, in turn, benefits the analysis of numerical relations [21, 26]. In Fig. 1, because the heap locations `&ar[x]` and `&ar[0]` *may be disjoint*, we assume their disjointness and exclude the program executions with $x = 0$ to allow strong updates at Line 7. Essentially, we compute an under-approximation as shown by the dashed rectangle in Fig. 2a.
- A summary-based compositional analysis [63] can be designed to significantly improve the efficiency by generating and reusing function summaries. Thanks to the disjointness, our summary is compact and does not need to encode the exponential number of possible aliasing patterns from the function's environment [23]. For instance, the summary of `set_mem` tracks states for the symbolic heap locations `&ar[x]` and `&ar[0]` independently, whose instantiation in `baz` precisely infers that `t1 == 1` and `t2 == 0`, avoiding producing false report for Line 13.

Analysis Design. At a high level, we perform a bottom-up, summary-based compositional analysis in the style of symbolic execution, exploiting the disjointness property to perform strong updates at symbolic heap locations. Specifically, we propose a symbolic memory model that incorporates numerical information in heap reasoning to identify must-alias pairs of symbolic heap locations, and assume two symbolic heap locations to be disjoint if they are not must-aliases. To coordinate the disjointness property with the compositional analysis, our function summary is predicated on the disjointness assumptions introduced during summary generation. The disjointness assumptions are carefully validated before applying the summary because they can be disproved by newly discovered aliasing at the call site. Formally, the analysis forms an



(a)

$$array = newSolverArray()$$
$$array' = store(array, 0, x)$$
$$array'' = store(array', x, 0)$$

(b)

Figure 2: (a) A disjunction of heap states for precisely analyzing set_mem. The shaded node denotes a summary heap location containing a value of either x or 0. The case split produces two distinct heap states depending on whether x equals 0, e.g., the heap stores the value x at the offset 0 and 0 at the offset x when $x \neq 0$. (b) Array theory constraints to implicitly encode the disjunctive heap states.

Type τ	::=	int $\mid$ ptr(τ)
Variables $V = V_I \cup V_P$		*Program locations* $Loc \supseteq Alloc$
Arithmetic expression e_a	::=	$c \mid v \in V_I \mid v_1$ op $v_2, v_i \in V_I$
Pointer expression e_p	::=	$v \in V_P \mid b + o, b \in V_P, o \in V_I$
	$\mid$	$alloc^l(\tau, v), v \in V_I, l \in Alloc$
Program P	::=	F^+
Func F	::=	*define* $f(v_1, ..., v_n) = \{S\}, v_i \in V$
Statement S	::=	$v := e_a \mid v := e_p$
	$\mid$	$*v := k \mid k := *v, k \in V, v \in V_P$
	$\mid$	$assume(v_1 < v_2), v_i \in V_I$
	$\mid$	$f^l(a_1, \cdots a_n), a_i \in V, l \in Loc$
	$\mid$	$S_1; S_2 \mid nondet(S_1, S_2)$

Figure 3: A simple programming language.

under-approximation of an over-approximation [32] that trades soundness for precision and efficiency (detailed in § 3- § 5).

We implement the idea in Cod and perform an extensive evaluation on real-world software. The result shows that Cod is both precise and scalable: 29 buffer overflow bugs are uncovered from large and well-known codebases such as PHP with a false positive rate of 37%, while projects of millions of lines of code can be successfully analyzed within four hours. Fifteen of the bugs detected by Cod have been confirmed by the developers, with three CVE IDs assigned. Compared with four state-of-the-art static analyzers (Ikos [7], Symbiotic [12], Clang Static Analyzer (CSA) [1] and Infer [9]), Cod achieves close to 3× precision improvement or up to 120× speedups since they either lead to a false positive rate of over 90% or cannot terminate after twelve hours. In summary, we make the following contributions:

- A symbolic memory model that leverages numerical information to distinguish heap locations and introduces the disjointness assumption, resolving the mutual dependence between the heap and numerical values in a precise and efficient manner.
- A path-sensitive and compositional static analysis algorithm for buffer overflow detection that is both precise and scalable.
- An implementation and evaluation of the idea to empirically demonstrate our improvement.

2 PRELIMINARIES

We formalize our approach using the language in Fig. 3. A variable v in the program is either an integer ($v \in V_I$) or a pointer ($v \in V_P$), whose type is denoted by $type(v)$. Arithmetic expression encodes numerical computations over integer variables. Array and dynamic memory allocation are uniformly represented by $alloc^l(\tau, v)$, indicating that an object with the size v is allocated at $l \in Alloc$. $b + o$ denotes a pointer arithmetic expression with base pointer b and numerical offset o. Heap operations are carried out through the load and store statements. Other statements include assumption, sequencing, non-deterministic choice (represented by $nondet$), and function calls (represented by $f^l(a_1, \cdots a_n)$, where $l \in Loc$ is the call site location). In this work, we assume that loops in the control flow graph and call graph are finitely unrolled, similar to previous works on static bug detection [2, 64, 69, 73].

In order to uniformly encode the numerical values and heap locations, we define *symbolic access path*, generalizing access path [66] by embedding numerical offsets:

Definition 2.1. (Symbolic Access Path / Symbolic Heap Location) A numeric value va is $va \in \Gamma ::= \hat{v} \in SymVar \mid va_1 \ \widehat{\text{op}}\ va_2$, where $SymVar$ is the set of symbolic variables (for convenience, assume any constant $c \in SymVar$) and $\widehat{\text{op}}$ corresponds to the binary operation used by the language in Fig. 3.

A symbolic access path is $\pi \in Acc ::= va \in \Gamma \mid l^+(sz) \mid par_i \mid \pi +$ off, off $\in \Gamma \mid * \pi$, where π can be of either integer (e.g., $va \in \Gamma$) or pointer type (in which case π is a symbolic heap location).

$l^+(sz)$ denotes a memory object with the context-sensitive allocation site l^+ ($l \in Alloc$) and size $sz \in \Gamma$, while par_i denotes the value of the i-th function parameter. More complex symbolic access path can be obtained by adding a numerical offset off $\in \Gamma$ or performing a dereference. We say π is degenerated if no variable in $SymVar$ appears in π, i.e., π involves no symbolic offset and can be identified with a conventional access path.

A symbolic heap location can refer to different locations under different program executions:

Definition 2.2. (Concretization of Symbolic Heap Location) Given the analysis state σ, $\gamma(\pi, \sigma) \in \wp(Acc)$ is the set of heap locations represented by π under σ, where all $\pi_0 \in \gamma(\pi, \sigma)$ are degenerated. We defer the formal definitions for σ and γ to § 4.

Example 2.1. Let π be the symbolic heap location corresponding to `&ar[x]` in Fig. 1. Analyzing the function `set_mem` alone, $\gamma(\pi, \sigma)$ denotes all the possible locations $\{ar, ar + 4, ar + 8, \ldots\}$ from the array (assuming sizeof(int) = 4). Meanwhile, analyzing `set_mem` with its only call site at Line 11, $\gamma(\pi, \sigma)$ denotes the single array element `&ar[1]` because the numerical state indicates that `x=1`.

3 OVERVIEW

This section describes how we use the disjointness assumptions to achieve precision and efficiency, as well as the challenges in establishing the disjointness property (§ 3.1). Then, we outline how we address these challenges in § 3.2 and § 3.3.

3.1 Problem Statement

Our analysis utilizes disjointness assumptions to precisely analyze a subset of the state space that permits strong updates. This helps to avoid the expensive case analyses normally required for precisely handling symbolic heap locations and also simplifies the design of the compositional analysis.

Running Example. We illustrate our approach using the example in Fig. 1, which consists of summary generation and application. As shown in Fig. 4, the summary of `mem_alloc` signifies that a new memory object is allocated at Line 2 (l_2), has allocation size of four times its first parameter ($4par_1$), and is returned to the caller (ret).

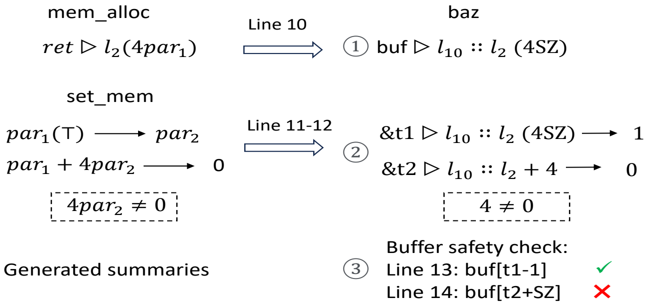

mem_alloc
$ret \triangleright l_2(4par_1)$
set_mem
$par_1(\top) \longrightarrow par_2$
$par_1 + 4par_2 \longrightarrow 0$
$4par_2 \neq 0$
Generated summaries
Line 10
Line 11-12
baz
① buf $\triangleright\ l_{10} :: l_2$ (4SZ)
② &t1 $\triangleright\ l_{10} :: l_2$ (4SZ) $\longrightarrow$ 1
&t2 $\triangleright\ l_{10} :: l_2 + 4 \longrightarrow$ 0
$4 \neq 0$
③ Buffer safety check:
Line 13: buf[t1-1] ✓
Line 14: buf[t2+SZ] ✗

Figure 4: Demonstration of Cod on the code of Fig. 1. The left part illustrates the generated summaries for mem_alloc and set_mem, and the right part shows how the analysis of baz applies the summaries. $e \triangleright \pi_1 \longrightarrow \pi_2$ denotes that the expression e contains the value of π_1 that points to π_2. Disjointness assumptions are marked with dashed rectangles.

Meanwhile, the summary of `set_mem` suggests that the memory object pointed by its first parameter has an unknown size (denoted by $par_1(\top)$), but stores par_2 at the offset zero and zero at the offset $4par_2$.

Notably, during the analysis of `set_mem`, we introduce the disjointness assumption $4par_2 \neq 0$, which allows strong updates to the heap locations `&ar[0]` (par_1) and `&ar[x]` ($par_1 + 4par_2$) at Line 7 and yields a precise summary. Also, the disjointness assumption frees us from enumerating possible aliasing patterns from the function's environment and makes the summary compact.

Next, when we analyze the function `baz`, we apply the summaries of `mem_alloc` and `set_mem` respectively at step ① and ② of Fig. 4: the analysis infers that the memory object pointed by `buf` is allocated by the call stack {Line 10, Line2} with size 4SZ ($l_{10} :: l_2(4\text{SZ})$), and stores 1 at the offset zero and zero at the offset 4. By enforcing disjointness and distinguishing symbolic heap locations, our analysis is able to correctly report the bug at Line 14 and avoid the false positive at Line 13 in step ③ of Fig. 4.

Challenges. The central problem of our analysis is to strengthen the analysis state σ_0 to σ by actively introducing disjointness assumptions such that the disjointness property holds at σ:

Definition 3.1. (Disjointness Property) Given the analysis state σ and the set of symbolic heap locations $\{\pi_1, \ldots, \pi_n\}$ tracked by σ, σ satisfies the disjointness property if $\forall i \neq j.\ \gamma(\pi_i, \sigma) \cap \gamma(\pi_j, \sigma) = \emptyset$. Apparently, storing to π_i does not change the state for π_j with the disjointness property, even though each may refer to many concrete heap locations. When the disjointness property holds, we say *strong updates* are allowed on $\{\pi_1, \ldots, \pi_n\}$ under σ.

Unfortunately, effectively establishing the disjointness property is non-trivial: It is hard to efficiently determine whether two symbolic heap locations may be disjoint such that their disjointness can be enforced. In particular, there are two main challenges:

Checking disjointness is costly. Deciding the disjointness between symbolic heap locations is difficult and can involve an unbounded number of heap locations. While we can encode the condition in Definition 3.1 as an extra constraint $\bigwedge_{i \neq j, 1 \leq i,j \leq n} \pi_i \neq \pi_j$, solving it would require analyzing various cases for the numerical offsets of $\{\pi_1, \ldots, \pi_n\}$. This contradicts our goal of using disjointness assumptions to avoid expensive case analyses. Additionally, the size of the extra constraint is quadratic to the number of symbolic heap locations, further impacting performance.

NumEnv	*AbsStore*
x ↦ $\hat{x}$	ar ↦ $\hat{x}$, ar + 4$\hat{x}$ ↦ 0

(a)

Before	*NumEnv*	After
ar + 4x	[x ↦ $\hat{x}$]	ar + 4$\hat{x}$
buf + 4(t1 − 1)	[t1 ↦ 1]	buf

(b)

Figure 5: (a) Our symbolic memory model built for `set_mem` of Fig. 1. *NumEnv* and *AbsStore* denote the numerical environment and the abstract store respectively. (b) On the fly construction of symbolic access paths based on the numerical environment, for `&ar[x]` and `&buf[t1-1]` in Fig. 1.

Inter-procedural disjointness assumption propagation is costly. In a compositional analysis, where each function is analyzed independently of its calling contexts, the disjointness assumptions introduced during summary generation need to be validated before applying the summary. These assumptions can either be refuted by newly discovered aliasing at the call site or further propagated into the caller's context. However, the validation overhead can be significant since the analysis needs to frequently apply summaries, and the number of possible aliasing patterns from a function's environment is exponential [23].

3.2 Approximating Disjointness Property

To check disjointness efficiently, our main idea is to leverage numerical information in heap reasoning to efficiently identify *must-alias pairs* of symbolic heap locations. We assume two symbolic heap locations to be disjoint only if they are not must-aliases.

Symbolic Memory Model for Identifying Must Aliases. It is challenging to determine when to introduce the disjointness assumption, as illustrated by the following example.

Example 3.1. Consider the following code snippet:

```
define f(n:int, x:int) = {p=alloc^{l1}(int, 8); y = x-1; ...}
```

It is reasonable to assume that `p+n` and `p+x` are disjoint since the values for `x` and `n` are unknown. However, it would be wrong to assume the disjointness between `p+x` and `p+y+1`, because the numerical relation `y=x-1` indicates that they must alias each other.

We propose a symbolic memory model that judiciously incorporates numerical information in heap reasoning. Specifically, we use the numerical environment, i.e., the mapping between numerical variables and their symbolic values, to compute symbolic access paths for heap locations, but ignore any constraint enforced by the branch conditions. Two heap locations are determined to be must-aliases if their corresponding symbolic access paths are *syntactically equivalent*, otherwise we assume they are disjoint. For instance, in Ex. 3.1, (`p+x`, `p+y+1`) are must-aliases for sharing the same symbolic access path $l_1 + 4\hat{x}$ assuming that `x` has the value $\hat{x} \in \Gamma$ in the numerical environment.

Example 3.2. Fig. 5a shows our symbolic memory model for the function `set_mem` of Fig. 1, where we efficiently establish the disjointness property. The symbolic heap locations ar + 4$\hat{x}$ and ar are assumed to be disjoint because they are syntactically different ($\hat{x}$ denotes the unknown symbolic value for the parameter `x`). Notice that we have performed strong updates to these heap locations to build a precise and compact abstract store.

Fig. 5b illustrates the construction of symbolic access paths for heap locations based on the numerical environment. For example, `&buf[t1-1]` (i.e., $\text{buf} + 4(\text{t1} - 1)$) is evaluated to `&buf[0]` under the numerical environment $[\text{t1} \mapsto 1]$ at Line 13 of the code, precisely resolving to the first concrete element in the array.

Trade-offs in the Approximation. We compare symbolic access paths to identify must aliases and shrink the program state space by introducing the disjointness assumptions. Our disjointness checking over-approximates branch conditions in the program with non-deterministic choices, which boosts the efficiency but can lead to falsely assumed disjointness. Meanwhile, introducing disjointness assumptions under-approximates the state space to enable strong updates and improves the precision, but can lead to missed bugs. In § 4, we formally demonstrate that the analysis corresponds to an under-approximation of an over-approximation [32] and further discuss about the trade-offs.

3.3 Propagating Interprocedural Disjointness

To coordinate the disjointness property with the compositional analysis, we carefully design the function summary to incorporate the disjointness assumptions, which we validate and propagate inter-procedurally during summary application.

Compact Function Summary. Our summary is generated from the symbolic memory model by projecting it to the parameter reachable symbolic heap locations. Utilizing symbolic access path, the summary uniformly represents both the heap locations and numerical variables. The disjointness assumptions introduced during summary generation serve as the precondition of the summary and free it from encoding possible aliasing patterns from the function's environment, making the summary compact.

Summary Application with Disjointness Propagation. At a call site, the precondition of the callee's summary is checked because newly discovered aliasings at the caller can disprove it and invalidate the summary. If the disjointness assumption is not refuted, it is further propagated to the caller and the summary is applied to instantiates the callee's effects on both the heap and value. Otherwise, the analysis degrades to a non-compositional mode and proceeds by "inlining" the callee's code.

We achieve efficiency in the summary application by employing the same procedure of disjointness checking as in § 3.2: At the function call site, the disjointness assumption is only refuted when the concerned symbolic heap locations used by the summary are found to be must-aliases in the current context. In this manner, we preserve the *under of over* approximation for the disjointness property in the modular analysis setting.

Example 3.3. The summary of `set_mem` (shown in Fig. 4) is generated from the symbolic memory model in Fig. 5a and predicated on the disjointness assumption $4par_2 \neq 0$. At the call site Line 11 in Fig. 1, the analysis attempts to apply the existing summary.

With the second parameter given the value 1, the disjointness assumption $4par_2 \neq 0$ is instantiated into $4 \neq 0$ and found to hold. Therefore, the summary of `set_mem` is applied: the two stored memory locations par_1 and $par_1 + 4par_2$ from the summary are instantiated to $l_{10} :: l_2$ and $l_{10} :: l_2 + 4$ respectively (par_1 instantiates to what `buf` points to, i.e., $l_{10} :: l_2$, c.f. Fig. 4).

4 PROGRAM ABSRACTION

In this section, we formally define our abstraction of the program state and the function summary, which are essential to approximating the disjointness property in a modular analysis.

Analysis State. We propose the following symbolic memory model to track the heap and numerical state simultaneously:

Definition 4.1. A program state is a four-tuple $\sigma \in State \overset{\text{def}}{=} NumEnv \times PtrEnv \times AbsStore \times Cond$, where

- $NumEnv \overset{\text{def}}{=} V_I \rightarrow \Gamma$ Numerical environment.
- $PtrEnv \overset{\text{def}}{=} V_P \rightarrow Acc$ Pointer environment.
- $AbsStore \subseteq Acc \times Acc$ Abstract store.
- $\varphi \in Cond ::= true \mid false \mid va_1 < va_2 \mid \varphi_1 \wedge \varphi_2$ Constraint.

The numerical state is decomposed into $NumEnv$ (recording the current symbolic values for numerical variables) and $Cond$ (the path condition for certain program execution). The heap state is characterized by $PtrEnv$ and $AbsStore$, which tracks the address contained in a pointer variable and the points-to target of the address.

Under σ, the set of degenerated heap locations (c.f. Definition 2.1) represented by a symbolic heap location can be defined as:

Definition 4.2. (Concretization of Symbolic Heap Location, instantiated from Definition 2.2) Given a symbolic heap location $\pi \in Acc$ and a mapping $M \in SymVar \rightarrow Integer$, let $concr(\pi, M) \overset{\text{def}}{=} \pi[M[x]/x]_{x \in dom(M)}$ where $\pi[M[x]/x]$ substitutes $M[x]$ for x in π. The concretization of π under a program state σ is $\gamma(\pi, \sigma) \overset{\text{def}}{=} \{concr(\pi, M) \mid M \models \sigma\}$, where $M \models \sigma$ indicates that M is a model of the constraint φ of σ, i.e., $\sigma = (\_, \_, \_, \varphi), M \models \varphi$[1].

We now rephrase the disjointness property:

Definition 4.3. (Disjointness Property, instantiated from Definition 3.1) Let $\underline{\text{disjoint}}(\pi_1, \pi_2, \sigma) \overset{\text{def}}{=} \gamma(\pi_1, \sigma) \cap \gamma(\pi_2, \sigma) = \emptyset$ denote the *must disjoint relation* between $\pi_1, \pi_2 \in Acc$ under σ. Given a state $\sigma = (\_, \_, store, \_)$ with $store \in AbsStore = (\pi_1, pts_1), \ldots, (\pi_k, pts_k)$, the disjointness property holds if $\forall i \neq j.\ \underline{\text{disjoint}}(\pi_i, \pi_j, \sigma)$.

Syntactical Approximation of the Disjointness Property. As illustrated in § 3, we efficiently distinguish among symbolic heap locations by leveraging the numerical environment to construct and syntactically compare symbolic access paths, which embeds the values of the numerical variables in its representation:

Example 4.1. Given a state $\sigma = ([m \mapsto 2, n \mapsto \hat{n}], [p \mapsto l_1], [l_1 \mapsto l_2, l_2 \mapsto 1], \_)$, the symbolic access path for $p + m + n$ is constructed as $\pi = (p + m + n)[l_1/p][2/m][\hat{n}/n] = l_1 + \hat{n} + 2$.

Whenever two heap locations have syntactically different symbolic access paths, the analysis assumes that they are disjoint and tracks their states independently in the abstract store. Formally, we ensure that our analysis state satisfies an uniqueness property:

Definition 4.4. (Uniqueness Property) Let $\equiv$ denote the equivalence relation of symbolic access paths (Definition 2.1), which is

[1] We use _ to denote a component in $State$ that is not used in our exposition.

defined based on comparing the syntax trees. Given a state $\sigma = (\_, \_, store, \_)$ with $store \in AbsStore = (\pi_1, pts_1), \ldots, (\pi_k, pts_k)$, σ satisfies the uniqueness property if $\forall . i \neq j : \pi_i \not\equiv \pi_j$, i.e., $AbsStore$ in Definition 4.1 is a function in $Acc \rightarrow Acc$.

The important question is, what is the relationship between the uniqueness property preserved by the analysis state and the disjointness property (Definition 4.3) that we aim for? The following theorem formally addresses the approximation:

Theorem 4.2. *(Approximation of the disjointness property) Let $\sigma[\varphi]$ be a state that is the same as σ except that the constraint is replaced by φ. Given σ satisfying the uniqueness property with $store = (\pi_1, pts_1), \ldots, (\pi_k, pts_k)$, we have $\forall i \neq j$: $\underline{disjoint}(\pi_i, \pi_j, \sigma[\varphi_d])$, where $\varphi_d \overset{def}{=} \bigwedge_{i \neq j} \pi_i \neq \pi_j$ and $\varphi_d \not\equiv false$.*

Theorem 4.2 has the following implications. First, our abstraction is sound for discovering must-aliases. Specifically, when two heap locations share the same representation of the symbolic access path under our memory model, they must be aliased to each other. Second, we may produce false positives for deciding disjointness due to ignoring numerical constraints. For instance, if the constraint in Ex. 4.1 were $-3 < \hat{n} < -1$, then l_1 and $l_1 + \hat{n} + 2$ must be aliases even though they are syntactically different.

Remark 4.1. Notice that the construction of symbolic access paths has already involved arithmetic rewriting based on the numerical environment (as shown by Ex. 4.1), as opposed to the syntactic access path used in RacerD [5]. Theoretically, our abstraction of the disjointness property forms an under-approximation of an over-approximation [32]. Particularly, we first identify must-aliases pairs of symbolic heap locations in the over-approximated program where branch conditions are replaced with non-deterministic choices. Then we perform an under-approximation of that over-approximation by strengthening the state to $\sigma[\varphi_d]$ as in Theorem 4.2, thereby only accounting for the program executions where the concerned symbolic heap locations are disjoint.

The *under-of-over* paradigm may lead to both false positives and false negatives. Similarly, when targeting realistic programs with millions of lines of code, existing static bug-finding tools are neither strict over- nor under-approximation [2, 3, 64, 74].

Function Summary. Our function summary tracks (1) the observable effects of the function and (2) a set of safety queries for the buffer accesses made within the function:

Definition 4.5. A function summary is a set of pairs $Smry \in FuncSmry \overset{\text{def}}{=} \wp(State \times \wp(Cond))$, where $(\sigma, Qs) \in Smry$ indicates that σ is the final state for a specific program execution of the function, and $Qs \in \wp(Cond)$ is a set of queries generated for that execution. Each query $Q \in Cond$ is a constraint encoding the condition for triggering buffer overflow at a certain dereference site. We assume a global summary environment $SmryEnv$ that maps from a function f to its summary.

Although we ignore the constraints when determining the disjointness relation among heap locations, they are recorded in Qs for building the conditions for triggering buffer overflow.

INIT-STATE

$$\frac{\text{Function definition} : define\ fun(f_1, ..., f_n) = \{S\} \\ e_0 = \lambda f_i \in V_I . \widehat{f_i} \qquad e_1 = \lambda f_j \in V_P . par_j \qquad st = \lambda par_j \in range(e_1).(*par_j)}{init\_abs(fun) \vdash (e_0, e_1, st, true)}$$

ARITH-NUM

$$\frac{aexp' = aexp[e(x)/x]_{x \in dom(e)}}{(e, \_, \_, \_) \vdash v := aexp : (e[v \mapsto aexp'], \_, \_, \_)}$$

ARITH-PTR

$$\frac{}{(e_0, e_1, \_, \_) \vdash v := b + o : (e_0, e_1[v \mapsto e_1(b) + e_0(o)], \_, \_)}$$

ALLOC

$$\frac{}{(e_0, e_1, \_, \_) \vdash v_1 := alloc^l(\tau, v_2) : (e_0, e_1[v_1 \mapsto l(e_0(v_2))], \_, \_)}$$

STORE

$$\frac{\pi = e_1(v) \qquad val = (type(k) = \text{int})\ ?\ e_0(k) : e_1(k) \qquad st' = st[\pi \mapsto val]}{(e_0, e_1, st, \_) \vdash *v = k : (e_0, e_1, st', \_)}$$

LD-INT

$$\frac{val = st(e_1(v)) \qquad type(k) = \text{int}}{(e_0, e_1, st, \_) \vdash k = *v : (e_0[k \mapsto val], e_1, st, \_)}$$

LD-PTR

$$\frac{val = st(e_1(v)) \\ type(k) = ptr(\tau) \qquad st' = (val \in dom(st))\ ?\ st : st[val \mapsto (*val)]}{(\_, e_1, st, \_) \vdash k = *v : (\_, e_1[k \mapsto val], st', \_)}$$

SMRY-GEN

$$\frac{\text{Function definition} : define\ fun(f_1, ..., f_n) = \{S\} \\ init\_abs(fun) \vdash \sigma_0 \qquad (\sigma_0, \emptyset) \vdash S : (\sigma_{fun}, Q)}{(\sigma_{fun}, Q) \in SmryEnv(fun)}$$

Figure 6: Intra-procedural inference rules. *assume*, *seq* and *nondet* are handled in standard manner (rules omitted).

5 ANALYSIS ALGORITHM

This section details our algorithm for buffer overflow detection. We first present our intra-procedural analysis, especially the rules to effectively handle memory operations at symbolic heap locations in § 5.1, and describe the application of function summaries to support inter-procedural reasoning in § 5.2. Finally, we illustrate the method for reporting buffer overflow bugs in § 5.3.

5.1 Intra-procedural Analysis

Our analysis preserves the uniqueness property of σ and utilizes it to handle memory operations at symbolic heap locations. Fig. 6 formulates the analysis using judgment $\sigma \vdash S : \sigma'$, meaning that executing S transforms the state σ to σ', which we elaborate below.

State Initialization. We analyze a function independently of its calling context where the initial values for function parameters are unknown. Rule *init-state* sets up the initial state and also initializes our disjointness assumptions: each integer parameter f_i is associated with an unknown symbolic value $\widehat{f_i}$ in e_0, each pointer parameter f_j points to an unknown memory object denoted by the access path par_j, and the initial store contains pairs of $(par_j, *par_j)$, implicitly assuming that different par_js do not alias.

Construction of Symbolic Access Path. As illustrated by Ex. 4.1, the construction of the symbolic access path leverages the numerical environment in arithmetic simplification. Rule *arith-ptr* updates the symbolic access path for the assigned pointer variable by adding the symbolic offset, $e_0(o)$, to the symbolic access path, $e_1(b)$, of the base pointer. Meanwhile, numerical computation among integer variables is handled by the Rule *arith-num*.

Handling Memory Operations. Heap locations come from not only memory allocations, e.g., Rule *alloc* uses $l(e_0(v_2))$ to indicate that the memory object allocated at the location l has size $e_0(v_2)$, but also function parameters. Thus, special treatments are needed to handle unknown memory objects from the function's environment. Moreover, our analysis should preserve the uniqueness property during the handling of memory operations, which is crucial for enabling strong updates to symbolic heap locations (Theorem 4.2).

Handling Unknown Points-to Target. To handle a load or store operation at v, we first obtain the symbolic access path $\pi = e_1(v)$ for v in the pointer environment, e_1, and subsequently retrieve the points-to target by looking up the abstract store st. However, π may never have been stored before, e.g., when it is derived from the function parameters and points to the unknown environment. Therefore, we make a distinction between loading an integer (Rule *ld-int*) and loading a pointer (Rule *ld-ptr*). A form of lazy initialization [38] is performed in the Rule *ld-ptr*: when the loaded value val is a pointer but has no points-to target, we initialize in st to record that val points to the unknown target $*val$.

Preserving the Uniqueness Property. When updating the abstract store st with $st[\pi \mapsto val]$ in Rule *store*, we first lookup for a pair $(\pi_0, val_0) \in st$ such that $\pi \equiv \pi_0$ (Recall that $\equiv$ is the syntactical equivalence relation of symbolic access paths). If found, val_0 is overwritten with val. Otherwise, a new pair (π, val) is added to st. Essentially, memory operations are carried out on symbolic access paths that are uniquely identified by their syntactical structures.

Example 5.1. Consider the analysis of the following function:

```
define f(p:ptr(ptr(int)), q:ptr(int), n:int) =
    {*q = n; t = *p; *t = 1; k=t+n; *k = 2;}
```

Utilizing Rule *init-state* of Fig. 6, we start from $\sigma_0 = (e_n, e_p, st_0, c)$ where $e_n = [n \mapsto \hat{n}], e_p = [p \mapsto par_1, q \mapsto par_2], st_0 = [par_1 \mapsto *par_1, par_2 \mapsto *par_2], c = true$. The analysis of f produces:

$$\begin{aligned}
\sigma_0 &\rightarrow_{store} & (e_n, e_p, st_1 = st_0[par_2 \mapsto \hat{n}], c)\\
&\rightarrow_{ld\text{-}ptr} & (e_n, e'_p = e_p[t \mapsto *par_1], st_2 = st_1[*par_1 \mapsto * * par_1], c)\\
&\rightarrow_{store} & (e_n, e'_p, st_3 = st_2[*par_1 \mapsto 1], c)\\
&\rightarrow_{arith\text{-}ptr} & (e_n, e''_p = e'_p[k \mapsto *par_1 + \hat{n}], st_3, c)\\
&\rightarrow_{store} & \sigma_f = (e_n, e''_p, st_4 = st_3[*par_1 + \hat{n} \mapsto 2], c)
\end{aligned}$$

Summary Generation. In order to generate function summary as defined in Definition 4.5, we extend the semantic judgment to $(\sigma, Q) \vdash S : (\sigma', Q')$ where $Q, Q' \in \wp(Cond)$ are the sets of generated queries before and after executing S, respectively (Rule *smry-gen* of Fig. 6). Recall that a query Q encodes the condition for triggering buffer overflow. For a dereferenced symbolic access path π, we extract both the numerical offset of π and the size of its pointed memory object to form the query condition. When the offset and size of π depend on the caller, we employ the uninterpreted functions $\overline{offset}(\pi)$ and $\overline{size}(\pi)$ that will be instantiated later during summary application.

Example 5.2. (Continuing Ex. 5.1) The summary generated for f is (σ_f, Q), where σ_f is defined in 5.1 and the queries in Q include $Q_1 = \overline{offset}(par_2) + 1 - \overline{size}(par_2) > 0$, $Q_2 = \overline{offset}(par_1) + 1 - \overline{size}(par_1) > 0$, $Q_3 = \overline{offset}(*par_1) + 1 - \overline{size}(*par_1) > 0$, $Q_4 = \overline{offset}(*par_1) + \hat{n} + 1 - \overline{size}(*par_1) > 0$. They correspond to the statements *q=n, t=*p, *t=1, and *k=2 respectively.

$$norm(\pi_0, st) \overset{\text{def}}{=} \begin{cases} st(\alpha) & \text{if } \pi_0 = *\pi \text{ and } norm(\pi, st) = \alpha, \alpha \in dom(st)\\ *\alpha & \text{elif } \pi_0 = *\pi \text{ and } norm(\pi, st) = \alpha, \alpha \notin dom(st)\\ \alpha + off & \text{elif } \pi_0 = \pi + off \text{ and } norm(\pi, st) = \alpha\\ \pi_0 & \text{otherwise} \end{cases}$$

INST-ACC

$$\frac{\begin{array}{c}\text{Callee definition} : define\ callee(f_1, \ldots, f_n) = S\\ \text{Call site} : callee^{l_0}(a_1, \ldots, a_n)\\ \text{Caller state at the call site} : \sigma_{caller} = (numE, ptrE, st, \_)\\ M = [par_i \mapsto ptrE(a_i)]_{f_i \in V_P} \cup [\widehat{f_j} \mapsto numE(a_j)]_{f_j \in V_I} \cup \lambda ls.l_0 :: ls\end{array}}{inst(\pi, \sigma_{caller}) = norm(map_\pi(\pi, M), st)}$$

SMRY-APP

$$\frac{\begin{array}{c}(\sigma_{callee}, Q) \in SmryEnv(callee) \qquad \sigma_{caller} = (numE, ptrE, st_1, \varphi_1)\\ \sigma_{callee} = (\_, \_, st_2, \varphi_2) \qquad st_2 = (\pi_1, tg_1), \ldots, (\pi_m, tg_m)\\ \forall i \neq j : inst(\pi_i, \sigma_{caller}) \not\equiv inst(\pi_j, \sigma_{caller})\\ st_{new} = st_1[inst(\pi_i, \sigma_{caller}) \mapsto inst(tg_i, \sigma_{caller})]_{1 \le i \le m}\\ \varphi_{new} = \varphi_1 \wedge inst(\varphi_2, \sigma_{caller})\\ Q' = inst(Q, \sigma_{caller}) \qquad \sigma'_{caller} = (numE, ptrE, st_{new}, \varphi_{new})\end{array}}{(\sigma_{caller}, Q_0) \vdash callee^{l_0}(a_1, \ldots, a_n) : (\sigma'_{caller}, Q_0 \cup Q')}$$

Figure 7: Summary application for inter-procedural analysis.

5.2 Inter-procedural Analysis

When processing function calls, our analysis will attempt to apply the existing summaries of the callee function (illustrated by Fig. 7). As the summary is predicated on the disjointness assumption introduced during the analysis, we check if the current context induces any must-alias pairs that invalidate the summary to preserve the uniqueness property inter-procedurally. If applicable, any observable effect encoded in the summary should be instantiated at the call site. Both aspects require instantiating the symbolic access paths used by the summary (Rule *inst-acc*), which we now elaborate.

Semantic Instantiation of Symbolic Access Path. The symbolic access path is first transformed syntactically into the caller's name space by $map_\pi(\pi, M) \overset{\text{def}}{=} \pi[M(x)/x]_{x \in dom(M)}$. Specifically, par_i is mapped to $ptrE(a_i)$ (the symbolic access path of the corresponding actual argument a_i), $\widehat{f_j}$ associated with the integer parameter f_j is mapped to its actual value $numE(a_j)$ at the call site, and any concrete location ls is prepended with the call site l_0 to achieve a context-sensitive heap abstraction [43, 67, 76].

The transformed symbolic access path $\pi_0 = map_\pi(\pi, M)$ may still contain sub expressions of the form $*\pi^p$, and the points-to target of π^p needs to be further resolved at the call site. This is due to the lazy initialization of the function's environment during summary generation, which can involve multiple levels of dereference through the function parameter. We thus apply $norm(\pi_0, st)$ to normalize π_0 under the caller's store st by eliminating the dereference operators in π_0 as many as possible. The *inst* procedure is naturally extended to instantiate constraints and queries (i.e., $\varphi \in Cond$) to the caller context by descending over their syntax trees.

Summary Application with Precondition Validation. Rule *smry-app* demonstrates the summary application process to handle function calls. The summary is applied only if the disjointness assumption is not refuted at the call site, i.e., the instantiation of the store st_2 of σ_{callee} should not contain two heap locations that are must-aliases (condition $inst(\pi_i, \sigma_{caller}) \not\equiv inst(\pi_j, \sigma_{caller})$).

Example 5.3. (Continuing Ex. 5.2) Consider the function:

```
define g(o1:ptr(int), o2:ptr(int)) = {
  o3=alloc$^{l_1}$(ptr(int), 16); *o3 = o2; x=4; f$^{l_2}$(o3, o1, x);}
```

The summary for the callee `f` is (σ_f, Q) defined in Ex. 5.2 where

$$\sigma_f = ([n \mapsto \hat{n}], [p \mapsto par_1, q \mapsto par_2, t \mapsto *par_1, k \mapsto *par_1 + \hat{n}],$$
$$[par_1 \mapsto *par_1, par_2 \mapsto \hat{n}, *par_1 \mapsto 1, *par_1 + \hat{n} \mapsto 2], true)$$

Let $\sigma_g = ([x \mapsto 4], [\mathrm{o1} \mapsto par'_1, \mathrm{o2} \mapsto par'_2, \mathrm{o3} \mapsto l_1], [l_1 \mapsto par'_2, par'_1 \mapsto *par'_1, par'_2 \mapsto *par'_2], true)$ be the analysis state at l_2, where par'_i denotes the ith parameter of g. Applying Rule *inst-acc*, we have $inst(par_1, \sigma_g) = l_1, inst(par_2, \sigma_g) = par'_1, inst(*par_1, \sigma_g) = par'_2$ and $inst(*par_1{+}\hat{n}, \sigma_g) = par'_2{+}4$. For instance, $*par_1$ is mapped to $*l_1$ and further normalized to par'_2 since l_1 points to par'_2 in σ_g. The summary application result is $\sigma_g, Q \vdash \mathrm{f}^{l_2}(\mathrm{o3}, \mathrm{o1}, \mathrm{x}) : \sigma'_g, Q'$, where $\sigma'_g = ([x \mapsto 4], [\mathrm{o1} \mapsto par'_1, \mathrm{o2} \mapsto par'_2, \mathrm{o3} \mapsto l_1], [l_1 \mapsto par'_2, par'_1 \mapsto 4, par'_2 \mapsto 1, par'_2 + 4 \mapsto 2], true)$.

Notice that Rule *smry-app* applies because symbolic access paths in the instantiated store are syntactically different from each other. Intuitively, we have propagated the disjointness assumptions to g, e.g., `o1` is assumed to be disjoint from `o2+4`, which will be further validated at call sites of g. The query Q_2 in Ex. 5.2 is resolved to be false because $Q'_2 = \overline{offset}(l_1) + 1 - \overline{size}(l_1) > 0 \equiv (0 + 1 - 16) > 0 \equiv false$, while Q'_1, Q'_3, Q'_4 still depend on callers of g.

When the summary is not applicable due to must-alias pairs discovered at the call site, the analysis conceptually "dives into" the body of the callee function, which we refer to as the on-demand inlining. To achieve this, we extend the analysis state defined in Definition 4.1 with an extra component of call stack that grows and shrinks accordingly during the calls and returns. Notice that after diving into the callee function, we may still apply the function summaries at the call sites inside it. Hence the benefits of the compositional design can still be enjoyed.

Remark 5.1. Modeling the unknown aliasing from a function's environment is a classic challenge in modular heap analysis. Most existing solutions fall into three classes: (1) Simply presume the "non-aliasing" among parameters in an unsound manner [25, 51, 74]; (2) Account for the exponential number of possible aliasing patterns when building the summary (e.g., the *relevant context inference* [16] and the approach in [23]), which is costly; and (3) Gradually construct the function summaries by reanalysis when new input patterns are discovered, such as the *partial transfer functions* [72].

However, neither of them distinguish among symbolic heap locations or summarize disjunctive heap states. Our "non-aliasing with check" approach records the disjointness assumption as the precondition of the function summary, which is capable of distinguishing among symbolic heap locations, but only describes an under-approximation of the function's behaviors.

5.3 Buffer Overflow Detection

The query conditions from the summary will be checked for reporting buffer overflow bugs. Suppose$(\sigma, Qs) \in SmryEnv(f)$, we report a buffer overflow for $Q \in Qs$ if Q is satisfiable as decided by an SMT solver and independent of unresolved caller dependencies (similar to the concept of "manifest bugs" in [44]). Specifically, Q should contain no symbolic access path of the form par_j or $\hat{f}_i$ introduced by the Rule *init-state* of Fig. 6. For instance, Q_2 in Ex. 5.2 can only be resolved to be safe during summary application at its caller (Ex. 5.3).

6 EVALUATION

We implement Cod based on LLVM [42]. Each loop in the control flow graph and call graph is replaced with an one-time unrolling of its body. Our evaluation aims to answer the following research questions:

- **RQ1**: How does Cod perform compared with other static analyzers in terms of buffer overflow detection?
- **RQ2**: The effectiveness of the important design choices in Cod.
  - **RQ2.1**: How does the disjointness assumption affect the precision and recall of the analysis?
  - **RQ2.2**: How often is the disjointness assumption invalidated during summary application?

To perform the evaluation, we have selected 12 real-life open-source C/C++ projects including *tmux*, *zstd*, *tcpdump*, *curl*, *redis*, *openssl*, *systemd*, *frr*, *php*, *binutils*, *qemu*, *gcc* (listed w.r.t the increasing order of program size), and assign them project IDs $\{0, \ldots, 11\}$. They range from a few thousand LoC to close to several million, cover diverse application domains, and are widely used and extensively checked by static analysis tools. We select the Juliet Test Suite with known ground truth to perform some of the control experiments.

All the experiments were performed on a computer with dual 20-core processors Intel(R) Xeon(R) CPU E5-2698 v4@2.20GHz and 256GB physical memory, running Ubuntu-20.04.

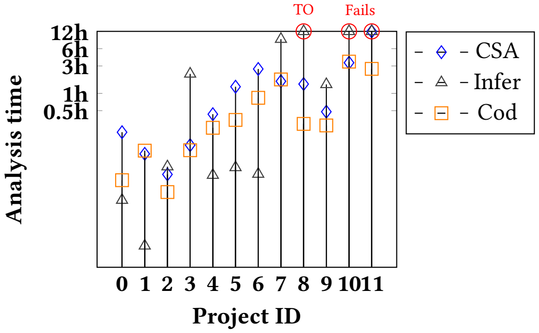
6.1 Effectiveness of Buffer Overflow Detection

We compare Cod with four state-of-the-art static analyzers: Ikos [7], Symbiotic [12], Clang Static Analyzer (CSA) [1] and Infer [9]. The first two are formal verification tools: Ikos is based on abstract interpretation and adopts a conservative memory model, while Symbiotic utilizes the array theory for handling memory operations at symbolic heap locations (c.f. § 1). Meanwhile, CSA and Infer share a similar design purpose with Cod by conducting "out-of-the-box" bug detection on large and realistic programs.

There are several challenges in making an end-to-end comparison against different static analyzers:

(1) Different tools may have different heuristics or engineering choices that may be undocumented or incomparable, e.g., Infer bounds the maximum number of disjunctions while CSA bounds the number of times a basic block is traversed.
(2) Subjectivity in reports classification: We may not be able to perfectly classify a bug report as false or true, e.g., when some global invariant of the code is unknown to a human analyst.

Due to these standard caveats in evaluating static analysis tools as well as the difference in design goals (verification vs bug finding), we only present a best-effort evaluation and do not assert that one tool is necessarily superior to another. For challenge (1), we adopt the default configurations of all the tools because they normally represent empirically good values. To check for buffer overflow, we run Infer with the option `--buffer-overrun-only` and run CSA with all buffer overflow related checkers. All the tools are run in a single-threaded setting. For challenge (2), we manually investigate the bug trace and its related code for each reported warning to determine the false positive rate. Further, we study the proportion of reports confirmed by the developers.



(a) Performance comparison over projects of increasing size.

ID	Cod	CSA	Infer	Symbiotic
	R / UR / TP	R / UR / TP	R / UR / TP	R / UR / TP
0	0 / 0 / 0	23 / 10 / 1	156* / 26 / 1	1 / 1/ 0
1	3 / 2 / 2	26 / 12 / 0	22 / 16 / 1	0 / 0 / 0
2	3 / 3 / 1	5 / 5 / 0	51 / 20 / 1	5 / 4 / 1
3	1 / 1 / 1	59 / 30 / 1	12 / 9 / 0	1 / 1 / 0
4	5 / 3 / 2	36 / 36 / 1	104* / 33 / 1	FAIL
5	11 / 3 / 1	77 / 52 / 2	983* / 41 / 1	0 / 0 / 0
6	7 / 4 / 2	716* / 65 / 1	774* / 49 / 1	37 / 13 / 1
7	15 / 7 / 5	74 / 50 / 1	716* / 53 / 0	3 / 2 / 0
8	16 / 6 / 4	169* / 62 / 2	TO	1 / 1 / 0
9	17 / 7 / 5	217* / 65 / 0	333* / 43 / 0	19 / 6 / 0
10	6 / 4 / 2	158* / 60 / 0	FAIL	FAIL
11	24 / 6 / 4	FAIL	FAIL	FAIL

(b) "#R": number of reports, "#UR": number of unique reports, "#TP": number of true positives. When a tool produces over 100 reports, we only examine 100 of them at random (marked with *).

Cod	CSA	Infer	Symbiotic	$\cap_1$	$\cap_2$	$\cap_3$
29 (15)	9 (4)	6 (1)	2 (0)	4 (4)	1 (1)	1 (1)

(c) Distribution of true positives. X(Y) indicates that X reports are classified as true positives and Y of them are confirmed by the developers. $\cap_1$, $\cap_2$, and $\cap_3$ denotes Cod ∩ CSA, Cod ∩ Infer and CSA ∩ Infer respectively, representing the true positives reported by multiple tools. The two true positives of Symbiotic are uniquely found.

Figure 8: Comparison of Cod, CSA, Infer, and Symbiotic. "TO" denotes timeout without producing reports. "FAIL" denotes analysis failure (due to crashes). Symbiotic doesn't finish within the timeout for all subjects.

Ikos and Symbiotic. Ikos successfully analyzes 11 % of the compiled bitcodes and fails on the rest because it only supports a subset of the LLVM instructions. In total, Ikos generates 2026 warnings. We randomly check 100 of them and discover that 72 reports are false positives, but could not determine for the rest because Ikos does not produce a bug trace to examine for the reported warnings.

Symbiotic is primarily designed for verifying memory safety. In our experiments, it does not finish the verification within 12 hours for all subjects, but is able to output the errors found up to the point when the timeout occurs. In total, 28 unique reports are generated with two true positives as shown in the last column of Fig. 8b. The relatively small number of found errors reflects the scalability issues of Symbiotic in large programs, as it employs heavyweight symbolic reasoning and a non-compositional design. Most of the false positives (24 out of 26) are due to Symbiotic conservatively reporting errors for accessing external memory objects.

Cod, CSA and Infer. CSA and Infer share similar design purposes with Cod for practical bug detection, which we now focus on comparing. The result is shown in Fig. 8. We make two basic observations. First, Cod has a false positive rate of 37% (= $\frac{\Sigma\#UR-\Sigma\#TP}{\Sigma\#UR} = \frac{46-29}{46}$), which is significantly more precise than CSA (98%) and Infer (98%). Notice that the buffer overflow checkers from CSA and Infer may generate a large number of reports, in which case we only examine 100 of them at random. While true bugs may be hidden in the reports not examined by us, the high false positive rate may discourage their adoptions by developers in realistic scenarios [3]. Second, Cod exhibits a runtime performance similar to that of CSA and Infer and scales to analyze millions of lines of code: it completes within around four hours across all projects.

The distribution of true positives detected is shown in Fig. 8c. As the data illustrates, different static bug finders tend to catch different bugs as also observed in [44]. Out of the 29 true positives detected by Cod, 15 bugs have been confirmed by the developers and three of them are severe enough to be assigned with CVE IDs. The bug reports can be found in https://tinyurl.com/ytktvd8v.

Answer to RQ1: Cod is the only static analyzer evaluated that both scales to million-line codebases and exhibits a relatively low false positive rate. Cod detects 29 buffer overflow bugs (15 confirmed) from real-world projects.

Discussions. To understand the high false positive rate of CSA and Infer and how distinguishing symbolic heap locations may help to improve it, consider the code snippet in Fig. 9a extracted from *curl*. CSA and Infer conservatively presume that the symbolic heap location `endofn = &ptr[nlen]` is an alias of `&ptr[0]` and generate a false positive for the dereference of `*endofn` (after the pointer decrement `endofn--`, an access at index −1 is deduced). In contrast, Cod utilizes the disjointness assumption to track the state of symbolic heap locations separately and avoids the false positive by precisely reasoning about the numerical value of the offset `nlen` (the code indeed checks `nlen` before the dereference).

As imprecision in static analysis accumulates and also originates from other aspects orthogonal to our work, it is difficult to give a quantitive measurement of how many false positives from CSA and infer are due to their inability to distinguish symbolic heap locations. Thus, we propose to further evaluate the effect of the disjointness assumption with ablation study in § 6.2.

Our treatment of loop, i.e., loop unrolling, is unsound and can lead to both false positives and false negatives. Fig. 9b shows an example where the loop invariant $sum = 2 \times i$ is crucial for discovering the off-by-one error when accessing `ar[sum]`. This is a common challenge for under-approximate static bug finders [2, 64, 69, 73] where "path dropping" is applied [54]. In contrast, Ikos and Symbiotic could in principle detect the bug by employing a relational numerical domain [19] or using the techniques in [13] respectively.

6.2 Effectiveness of the Design Choices

In this section, we examine the impact of the disjointness assumption and its preservation across function boundaries.

Effectiveness of the disjointness assumption, To study RQ 2.1, we implement a variant of our analysis Cod^{-} that does not distinguish among symbolic heap locations. In Cod^{-} we take the conventional layered design: a conservative pointer analysis is first

```
struct Cookie *Curl_cookie_add(...) {
    const char *endofn = &ptr[ nlen ];
    if(nlen >= MAX_NAME−1)  return 0;
    if(nlen) {
        endofn−−;
        if(ISBLANK(*endofn)) ...
    }
}
```

(a) Code extracted from *curl*.

```
int loop(...) {
    int sum = 0;
    int *ar = malloc(sizeof(int) * 2n)
    for (int i = 0; i < n; ++i) {
        sum += 2;
    }
    return ar[sum];
}
```

(b) A challenging loop program.

Figure 9: Case studies for strength and weakness of Cod.

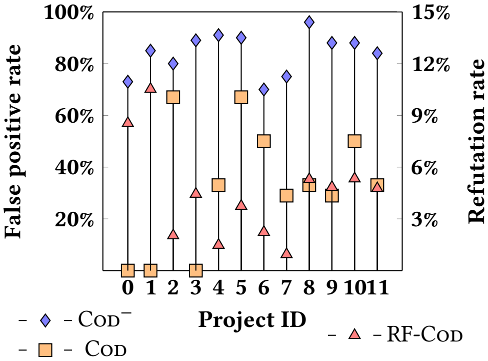


Figure 10: Study of the design choices. Legends Cod$^-$ and Cod: the false positive rates of the two variants (left y axis). RF-Cod: the proportion of times that the disjointness assumption is refuted at call sites for Cod (right y axis).

carried out (we use a flow-sensitive pointer analysis [49]), and the path-sensitive checking for buffer overflow utilizes the pointer analysis result to resolve memory dependencies. Cod$^-$ is unable to differentiate symbolic heap locations due to its reliance on the pointer analysis and loading from a symbolic heap location may return any value suggested by the pointer analysis non-deterministically. Other aspects of Cod$^-$ are the same as Cod, e.g., it also conducts a bottom-up symbolic execution to detect potential buffer overflows.

We evaluate Cod$^-$ on the same set of 12 projects and compare its precision with Cod. The result in Fig. 10 shows that Cod$^-$ produces significantly more false positives (FP rate 84%). We also compare Cod and Cod$^-$ using the Juliet Test Suite, where each test case comprises of X true bugs and Y safe variants. We have found that:

(1) Out of the 14164 test cases from five CWE categories related to buffer overflow (CWE number 121, 122, 124, 126, 127), Cod handles 9963 (70%) of them with 100% precision and recall. The rest includes 3952 test cases designed for the wide-character string (not supported by our implementation), 126 test cases related to the reasoning of string contents, and 123 test cases where the bugs do not manifest at the LLVM level (e.g., bugs that happen only if `sizeof(int *) == 4`). Interestingly, we have not found a false negative case due to our intentional shrinking of the state space by enforcing the disjointness assumption.

(2) Running Cod$^-$ on the 9963 test cases successfully handled by Cod, the false positive rate jumps drastically from 0% to 57%.

> Answer to RQ2.1: The disjointness assumption significantly improves the precision of Cod over Cod$^-$: false positive rate 37% vs. 84% on real projects and 0% vs. 57% on the Juliet Test Suite. Cod achieves 70% of recall on the Juliet Test Suite, providing confidence for its bug detection capability empirically.

Effectiveness of the compositional analysis. In our analysis, we check before summary application whether the disjointness assumption made by the summary (i.e., its precondition) is disproved at the call site. To study RQ 2.2, we postulate that the disjointness assumption holds most of the time, enabling the analysis to benefit from the disjointness for better precision and efficiency. Fig. 10 shows the experimental data for validating the disjointness assumption, from which we can conclude:

> Answer to RQ2.2: On average, the disjointness assumption is only disproved for 4.51% of the time, allowing the analysis to benefit from it for the majority of the cases.

7 RELATED WORK

Static buffer overflow detection / memory safety verification. Several static bug finders have been developed to detect buffer overflows [28, 45, 48, 51, 71, 75]. Meanwhile, methods based on symbolic model checking [34, 57] or abstract interpretation [6, 53] can be used to prove memory safety properties, such as the absence of buffer overflows. As discussed in § 1, these methods either adopt a conservative memory model that lacks precision in reasoning about disjunctive heap states based on numerical information or use heavyweight symbolic methods that lead to state space explosion. In contrast, Cod introduces disjointness assumptions into its memory model and enables strong updates to symbolic heap locations, improving both the precision and scalability of the analysis.

Combination of heap and numerical Analyses. Heap and numerical analysis can be carried out together by utilizing a product domain [6, 46, 53], or modularly combined without tampering with the constituents [14, 26, 27]. However, these works do not effectively utilize numerical information to enable strong updates in heap reasoning for better precision. McCloskey et al. [52] introduce a common predicate language to exchange facts between heap and numeric domains. However, their method is challenging to automate because it requires the user to provide shared predicates, such as a loop invariant. Works on the shape and static array analysis [11, 18, 31, 36, 50] have proposed many strategies to partition the unbounded heap. For instance, Gopan et al. [31] isolates individual heap cells to perform strong updates and constructs explicit partitions of the heap. However, these partition-based methods are susceptible to case explosion [21] in static analysis. In contrast, we do not infer invariants but instead target a subset of the program state space where the disjointness property holds, effectively tracking the states for symbolic heap locations for bug detection.

Refutation based refinement of heap analysis. A conservative heap analysis can be refined iteratively by on-demand backward analyses at selective program locations to improve the precision (as in checking memory leak [4] and JavaScript property accesses [68]), but may suffer from performance issues and fail due to timeout [4]. To statically detect buffer overflow based on refinement, we suspect that our disjointness assumptions and compositional design are still necessary to mitigate the state space explosion problem and be practical in large codebases.

Disjunctive static analysis. Maintaining a disjunction of states is related to sensitivity in static analysis [39, 59] and is crucial for the precision [1, 20, 44, 47, 64]. To balance the performance, existing works either merge abstract states to maintain a sound over-approximation [15, 47, 61] or drop disjuncts [1, 9, 40], e.g., state selection heuristics are proposed to maximize the number of detected bugs [40]. We select disjunctive states to exploit the disjointness property for enabling strong updates and improving the precision of buffer overflow checking.

Compositional heap analysis. Reasoning about parameter-induced aliasing has long been a major obstacle for modular heap analysis, which we have surveyed and compared in in Remark 5.1. In compositional shape analysis [10, 33], bi-abduction is used to infer function specifications expressed in terms of separation logic formulas [58], where a form of disjointness over the input shapes of the function is presumed implicitly and the summarization of the numerical state is left unspecified. In contrast, Cod explicitly introduces the disjointness assumption in its memory model and encodes both the heap and value state in the function summary.

Seahorn [34] introduces logical predicates to model the behavior of functions and uses an extensional theory of arrays to encode heap and numerical properties [35]. During constraint solving, sophisticated algorithms exploiting summaries are used and can lead to exponential cost [35, 41], which is too costly for our scenario.

8 CONCLUSION

We have presented Cod, a precise and scalable static buffer overflow detector, which introduces the heap disjointness assumptions to enable efficient strong updates to symbolic heap locations, and utilizes precise and compact function summaries for compositional analysis. Cod uncovered 29 buffer overflow bugs from large, real-world software efficiently with a low false positive rate.

9 DATA AVAILABILITY

The artifact of Cod is available at https://tinyurl.com/5349ndc9.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments on this work. This work is supported by the PRP/004/21FX grant from the Hong Kong Innovation and Technology Commission and research grants from Huawei and TCL. Peisen Yao is the corresponding author.

REFERENCES

[1] 2022. The Clang Static Analyzer. https://clang-analyzer.llvm.org/. Online; accessed 28-July-2022.

[2] Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and Precise Extended Static Checking. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 211–220. https://doi.org/10.1145/1368088.1368118

[3] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[4] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2491956.2462186

[5] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (oct 2018), 28 pages. https://doi.org/10.1145/3276514

[6] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 196–207. https://doi.org/10.1145/781131.781153

[7] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. 2014. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In *Software Engineering and Formal Methods*, Dimitra Giannakopoulou and Gwen Salaün (Eds.). Springer International Publishing, Cham, 271–277.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[9] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 459–465.

[10] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional Shape Analysis by Means of Bi-Abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/1480881.1480917

[11] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. 2017. Verifying array manipulating programs by tiling. In *International Static Analysis Symposium*. Springer, 428–449.

[12] Marek Chalupa, Vincent Mihalkovič, Anna Řechtáčková, Lukáš Zaoral, and Jan Strejček. 2022. Symbiotic 9: String Analysis and Backward Symbolic Execution with Loop Folding. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 462–467.

[13] Marek Chalupa and Jan Strejček. 2021. Backward Symbolic Execution with Loop Folding. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 49–76. https://doi.org/10.1007/978-3-030-88806-0_3

[14] Bor-Yuh Evan Chang and Xavier Rival. 2013. Modular Construction of Shape-Numeric Analyzers. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013 (EPTCS, Vol. 129)*, Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff (Eds.). 161–185. https://doi.org/10.4204/EPTCS.129.11

[15] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. 2007. Shape Analysis with Structural Invariant Checkers. In *Static Analysis*, Hanne Riis Nielson and Gilberto Filé (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 384–401.

[16] Ramkrishna Chatterjee, Barbara G Ryder, and William A Landi. 1999. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 133–146.

[17] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[18] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. *SIGPLAN Not.* 46, 1 (jan 2011), 105–118. https://doi.org/10.1145/1925844.1926399

[19] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) *(POPL '78)*. Association for Computing Machinery, New York, NY, USA, 84–96. https://doi.org/10.1145/512760.512770

[20] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, Complete and Scalable Path-Sensitive Analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 270–280. https://doi.org/10.1145/1375581.1375615

[21] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Fluid Updates: Beyond Strong vs. Weak Updates. In *Proceedings of the 19th European Conference on Programming Languages and Systems* (Paphos, Cyprus) *(ESOP'10)*. Springer-Verlag, Berlin, Heidelberg, 246–266.

[22] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Symbolic heap abstraction with demand-driven axiomatization of memory invariants. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 397–410.

[23] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 567–577.

[24] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.

[25] Manuel Fähndrich and Francesco Logozzo. 2011. Static Contract Checking with Abstract Interpretation. In *Formal Verification of Object-Oriented Software*, Bernhard Beckert and Claude Marché (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 10–30.

[26] Pietro Ferrara. 2014. Generic Combination of Heap and Value Analyses in Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 302–321.

[27] Zhoulai Fu. 2014. Modularly Combining Numeric Abstract Domains with Points-to Analysis, and a Scalable Static Numeric Analyzer for Java. In *Verification, Model Checking, and Abstract Interpretation*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 282–301.

[28] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. 2003. Buffer Overrun Detection Using Linear Programming and Static Analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (Washington D.C., USA) *(CCS '03)*. Association for Computing Machinery, New York, NY, USA, 345–354. https://doi.org/10.1145/948109.948155

[29] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification*, Werner Damm and Holger Hermanns (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 519–531.

[30] Isabel Garcia-Contreras, Arie Gurfinkel, and Jorge A. Navas. 2022. Efficient Modular SMT-Based Model Checking of Pointer Programs. In *Static Analysis*, Gagandeep Singh and Caterina Urban (Eds.). Springer Nature Switzerland, Cham, 227–246.

[31] Denis Gopan, Thomas Reps, and Mooly Sagiv. 2005. A Framework for Numeric Analysis of Array Operations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 338–350. https://doi.org/10.1145/1040305.1040333

[32] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. *Proc. ACM Program. Lang.* 3, POPL, Article 57 (jan 2019), 29 pages. https://doi.org/10.1145/3290370

[33] Bhargav S Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V Nori. 2009. Bottom-up shape analysis. In *International Static Analysis Symposium*. Springer, 188–204.

[34] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 343–361.

[35] Arie Gurfinkel and Jorge A Navas. 2019. Automatic Program Verification with SEAHORN. In *Engineering Secure and Dependable Software Systems*. IOS Press, 83–111.

[36] Nicolas Halbwachs and Mathias Péron. 2008. Discovering Properties about Arrays in Simple Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 339–348. https://doi.org/10.1145/1375581.1375623

[37] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2, Article 29 (jun 2016), 47 pages. https://doi.org/10.1145/2931098

[38] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Warsaw, Poland) *(TACAS'03)*. Springer-Verlag, Berlin, Heidelberg, 553–568.

[39] Se-Won Kim, Xavier Rival, and Sukyoung Ryu. 2018. A Theoretical Foundation of Sensitivity in an Abstract Interpretation Framework. *ACM Trans. Program. Lang. Syst.* 40, 3, Article 13 (aug 2018), 44 pages. https://doi.org/10.1145/3230624

[40] Yoonseok Ko and Hakjoo Oh. 2023. Learning to Boost Disjunctive Static Bug-Finders. ICSE.

[41] Anvesh Komuravelli, Nikolaj Bjorner, Arie Gurfinkel, and Kenneth L. Mcmillan. 2015. Compositional verification of procedural programs using horn clauses over integers and arrays. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. 89–96. https://doi.org/10.1109/FMCAD.2015.7542257

[42] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[43] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World. *SIGPLAN Not.* 42, 6 (jun 2007), 278–289. https://doi.org/10.1145/1273442.1250766

[44] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (apr 2022), 27 pages. https://doi.org/10.1145/3527325

[45] Wei Le and Mary Lou Soffa. 2008. Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 272–282. https://doi.org/10.1145/1453101.1453137

[46] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*. Citeseer, 96.

[47] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. 2017. Semantic-Directed Clumping of Disjunctive Abstract States. *SIGPLAN Not.* 52, 1 (jan 2017), 32–45. https://doi.org/10.1145/3093333.3009881

[48] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2010. Practical and Effective Symbolic Analysis for Buffer Overflow Detection. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 317–326. https://doi.org/10.1145/1882291.1882338

[49] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the Performance of Flow-Sensitive Points-to Analysis Using Value Flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 343–353. https://doi.org/10.1145/2025113.2025160

[50] Jiangchao Liu and Xavier Rival. 2015. Abstraction of arrays based on non contiguous partitions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 282–299.

[51] V Benjamin Livshits and Monica S Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. 317–326.

[52] Bill McCloskey, Thomas Reps, and Mooly Sagiv. 2010. Statically Inferring Complex Heap, Array, and Numeric Invariants. In *Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 71–99.

[53] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and Implementation of Sparse Global Analyses for C-like Languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 229–238. https://doi.org/10.1145/2254064.2254092

[54] Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (dec 2019), 32 pages. https://doi.org/10.1145/3371078

[55] OpenSSL. 2022. OpenSSL Security Advisory [01 November 2022]. https://www.openssl.org/news/secadv/20221101.txt. Online; accessed 7-Nov-2022.

[56] David M Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 68–78.

[57] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 106–113.

[58] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[59] Xavier Rival and Laurent Mauborgne. 2007. The Trace Partitioning Abstract Domain. *ACM Trans. Program. Lang. Syst.* 29, 5 (Aug. 2007), 26–es. https://doi.org/10.1145/1275497.1275501

[60] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1998. Solving Shape-Analysis Problems in Languages with Destructive Updating. *ACM Trans. Program. Lang. Syst.* 20, 1 (jan 1998), 1–50. https://doi.org/10.1145/271510.271517

[61] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (May 2002), 217–298.

[62] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.

[63] M Sharir and A Pnueli. 1978. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY. https://cds.cern.ch/record/120118

[64] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. 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* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 693–706. https://doi.org/10.1145/3192366.3192418

[65] Ziqi Shuai, Zhenbang Chen, Yufeng Zhang, Jun Sun, and Ji Wang. 2021. Type and Interval Aware Array Constraint Solving for Symbolic Execution. In *Proceedings of*

the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 361–373. https://doi.org/10.1145/3460319.3464826

[66] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (apr 2015), 1–69. https://doi.org/10.1561/2500000014

[67] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 387–400.

[68] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static Analysis with Demand-Driven Value Refinement. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 140 (oct 2019), 29 pages. https://doi.org/10.1145/3360566

[69] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 254–264. https://doi.org/10.1145/2338965.2336784

[70] Synopsys. 2020. The heartbleed bug. https://heartbleed.com/. Online; accessed 7-Nov-2022.

[71] David A Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. 2000. A first step towards automated detection of buffer overrun vulnerabilities.. In *NDSS*, Vol. 20. 0.

[72] Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-Sensitive Pointer Analysis for C Programs. *SIGPLAN Not.* 30, 6 (jun 1995), 1–12. https://doi.org/10.1145/223428.207111

[73] Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 351–363. https://doi.org/10.1145/1040305.1040334

[74] Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 351–363. https://doi.org/10.1145/1040305.1040334

[75] Yichen Xie, Andy Chou, and Dawson Engler. 2003. ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Helsinki, Finland) *(ESEC/FSE-11)*. Association for Computing Machinery, New York, NY, USA, 327–336. https://doi.org/10.1145/940071.940115

[76] Guoqing Xu and Atanas Rountev. 2008. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 225–236.

[77] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A Data-Driven CHC Solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 707–721. https://doi.org/10.1145/3192366.3192416

Received 15-DEC-2023; accepted 2024-03-02