# Per-Dereference Verification of Temporal Heap Safety via Adaptive Context-Sensitive Analysis

Hua Yan[1,4,5], Shiping Chen[2], Yulei Sui[3], Yueqian Zhang[1,2], Changwei Zou[1], and Jingling Xue[1]

[1] University of New South Wales, Australia
[2] Data61, CSIRO, Australia
[3] University of Technology Sydney, Australia
[4] Sangfor Technologies Inc., China
[5] Shenzhen Institutes of Advanced Technology, CAS, China

**Abstract.** We address the problem of verifying the temporal safety of heap memory at each pointer dereference. Our whole-program analysis approach is undertaken from the perspective of pointer analysis, allowing us to leverage the advantages of and advances in pointer analysis to improve precision and scalability. A dereference $\omega$, say, via pointer $q$ is unsafe iff there exists a deallocation $\psi$, say, via pointer $p$ such that on a control-flow path $\rho$, $p$ aliases with $q$ (with both pointing to an object $o$ representing an allocation), denoted $\mathcal{A}_\omega^\psi(\rho)$, and $\psi$ reaches $\omega$ on $\rho$ via control flow, denoted $\mathcal{R}_\omega^\psi(\rho)$. Applying directly any existing pointer analysis, which is typically solved separately with an associated control-flow reachability analysis, will render such verification highly imprecise, since $\exists\rho.\mathcal{A}_\omega^\psi(\rho) \wedge \exists\rho.\mathcal{R}_\omega^\psi(\rho) \not\Rightarrow \exists\rho.\mathcal{A}_\omega^\psi(\rho) \wedge \mathcal{R}_\omega^\psi(\rho)$ (i.e., $\exists$ does not distribute over $\wedge$). For precision, we solve $\exists\rho.\mathcal{A}_\omega^\psi(\rho) \wedge \mathcal{R}_\omega^\psi(\rho)$, with a control-flow path $\rho$ containing an allocation $o$, a deallocation $\psi$ and a dereference $\omega$ abstracted by a tuple of three contexts $(c_o, c_\psi, c_\omega)$. For scalability, a demand-driven full context-sensitive (modulo recursion) pointer analysis, which operates on pre-computed def-use chains with adaptive context-sensitivity, is used to infer $(c_o, c_\psi, c_\omega)$, without losing soundness or precision. Our evaluation shows that our approach can successfully verify the safety of 81.3% (or $\frac{93{,}141}{114{,}508}$) of all the dereferences in a set of ten C programs totalling 1,166 KLOC.

## 1 Introduction

Unmanaged programming languages such as C/C++ still remain irreplaceable in developing performance-critical systems such as operating systems, databases and web browsers. Such languages, however, suffer from memory safety issues. While spatial errors (e.g., buffer overflows) result in disastrous consequences (e.g., crashes, data corruption, information leakage, privilege escalation and control-flow hijacking), their temporal counterparts have also been shown to be equally deadly [54,28]. In particular, verifying absence of dangling pointer dereferences, an important temporal heap safety (referred to TH-safety hereafter), is thus desirable.

A quite flourishing research thread focuses on separation logic [42,15,59], which enables precise shape analysis for pointer-based data structures. Much research effort has been devoted to improving scalability and automation of separation-logic-based verification [58,13]. In particular, bi-abduction [8] empowers separation-logic-based verification to generate program specifications automatically for large programs with millions of lines of code, in a compositional manner rather than as a whole-program analysis. However, one of its inevitable downsides (from the perspective of whole-program analysis) is the loss of precision due to a maximum size limit imposed on disjunctions of pre-conditions manipulated in order to improve performance [8,7].

Memory errors can also be found by other techniques, such as data-flow analysis [41,14] and model checking [24,26,38]. Notably, pointer analysis [31,25,45,62,47,49] has recently made significant strides, providing a solid foundation for developing many pointer-analysis-based static analyses for detecting memory errors [9,30,44,57,56,60]. In this paper, we present a fully-automated pointer-analysis-based approach, called D$^3$ (a **D**isprover of **D**angling pointer **D**ereferences), to verifying absence of (i.e., disproving presence of) dangling pointers on a per dereference basis. Compared to separation-logic-based approaches, our approach tackles this verification task from a different angle. Instead of focusing on reasoning about a variety of pointer-based data structures precisely in separation logic, we focus on reasoning about pointer aliasing and control-flow reachability context-sensitively in a whole-program setting *on-demand*.

***Challenges*** We highlight three challenges, from the perspective of pointer analysis:

*Challenge 1: Modeling the Triple Troublemakers.* A TH-safety violation involves three distinct program locations, an allocation $o$ (representing an allocation site), a deallocation $\psi$ and a dereference $\omega$ , which must be all modelled precisely.

*Challenge 2: Resolving Aliases.* A dereference $\omega$ (via pointer $q$) is unsafe iff there exists a deallocation $\psi$ (via pointer $p$) such that on a control-flow path $\rho$, $p$ aliases with $q$ (with both pointing to an object $o$), denoted $\mathcal{A}_\omega^\psi(\rho)$, and $\psi$ reaches $\omega$ on $\rho$ via control flow, denoted $\mathcal{R}_\omega^\psi(\rho)$. Pointer aliasing, a well-known difficult static analysis problem, must be solved to guarantee both soundness and precision scalably for large programs. For the TH-safety verification, this is particularly challenging. Any existing $k$-limited context-sensitive pointer analysis that scales for large programs [45,25] (where $k \leq 3$ currently) is not precise enough (as $o$, $\psi$ and $\omega$ can often span across more than three functions). In addition, off-the-shelf pointer analyses provide the alias information between $\psi$ and $\omega$ but are oblivious to the control-flow reachability information from $\psi$ to $\omega$ (even if solved flow-sensitively), causing potentially a significant precision loss, since $\exists\rho.\mathcal{A}_\omega^\psi(\rho) \wedge \exists\rho.\mathcal{R}_\omega^\psi(\rho) \not\Rightarrow \exists\rho.\mathcal{A}_\omega^\psi(\rho) \wedge \mathcal{R}_\omega^\psi(\rho)$ (i.e., $\exists$ does not distribute over $\wedge$). Thus, increasing precision in our verification task requires pointer analysis to be not only more precise (with longer calling-contexts) but also synergistic with control-flow reachability analysis.

*Challenge 3: Pruning the Search Space.* To achieve high precision, a fine abstraction of control-flow paths (e.g., with adequate context-sensitivity) is required, but at a risk for causing path explosion. Furthermore, the presence of a large number of deallocation-dereference $(\psi, \omega)$ pairs that need to be checked further exacerbates the problem. Pruning the search space without any loss of precision is essential.

***Our Solution*** In this paper, we present a whole-program analysis approach that verifies TH-safety for each dereference $\omega$. Specifically, $\omega$ is considered safe iff there exists no deallocation $\psi$ such that the pair $(\psi, \omega)$ causes a dangling pointer dereference at $\omega$.

To meet Challenge 1, we model this verification problem context-sensitively with three contexts. We identify an allocation $o$, a deallocation $\psi$ (via pointer $p$) and a dereference $\omega$ (via pointer $q$) by a context tuple $(c_o, c_\psi, c_\omega)$ so that $\langle\!\langle c_o, o\rangle\!\rangle$ represents a context-sensitive heap object, i.e, an object $o$ created under $c_o$, $(c_\psi, p)$ deallocates what is pointed to by $p$ under $c_\psi$, and $(\omega, q)$ dereferences pointer $q$ under context $c_\omega$. We verify TH-safety with respect to $(o, \psi, \omega)$ by disproving the presence of a control-flow path that contains a context tuple, $(c_o, c_\psi, c_\omega)$, such that $\langle\!\langle c_o, o\rangle\!\rangle$, once deallocated at $(c_\psi, p)$, is still accessed subsequently at $(c_\omega, q)$ along the path.

To meet Challenge 2, we introduce a demand-driven pointer analysis that automatically infers the context information in pointer aliases so that the resulting alias analysis can correlate with an associated control-flow reachability analysis as required. Given a pointer $p$ at a deallocation (resp. a pointer $q$ at a dereference) without any context given, our pointer analysis will infer a context $c_\psi$ (resp. $c_\omega$), together with a context-sensitive object $\langle\!\langle c_o, o\rangle\!\rangle$, such that the context-sensitive pointer $(c_\psi, p)$ (resp. $(c_\omega, q)$) points to $\langle\!\langle c_o, o\rangle\!\rangle$, implying that $\exists\rho.\mathcal{A}_\omega^\psi(\rho)$. In addition, $c_\psi$ and $c_\omega$ are also required to satisfy the control-flow reachability constraint $\exists\rho.\mathcal{R}_\omega^\psi(\rho)$ simultaneously so that $\exists\rho.\mathcal{A}_\omega^\psi(\rho) \wedge \mathcal{R}_\omega^\psi(\rho)$ holds. This avoids false positives that satisfy $\mathcal{R}_\omega^\psi$ and $\mathcal{A}_\omega^\psi$ only for two distinct paths, respectively, which happens when $\exists\rho.\mathcal{A}_\omega^\psi(\rho) \wedge \exists\rho.\mathcal{R}_\omega^\psi(\rho) \not\Rightarrow \exists\rho.\mathcal{A}_\omega^\psi(\rho) \wedge \mathcal{R}_\omega^\psi(\rho)$. Finally, points-to queries are raised on-demand by traversing precomputed def-use chains (in order to improve efficiency) and by supporting full context-sensitivity (modulo recursion) to transcend $k$-limiting (in order to improve precision).

To meet Challenge 3, we make our context-sensitive analysis adaptive. A context tuple $(c_o, c_\psi, c_\omega)$ is reduced to $(c_o^{'}, c_\psi^{'}, c_\omega^{'})$ if $c_o$, $c_\psi$ and $c_\omega$ share a common prefix $c_{pre}$, so that $c_o{=}cons(c_{pre}, c_o^{'})$, $c_\psi{=}cons(c_{pre}, c_\psi^{'})$, and $c_\omega{=}cons(c_{pre}, c_\omega^{'})$, where $cons$ denotes string concatenation. This adaptive analysis aims to reduce exponentially many prefixes starting from `main()`, which would otherwise significantly impede scalability.

***Contributions*** This paper makes the following main contributions:

- We propose a fully automated approach to TH-safety verification on a per dereference basis, with a precise context-sensitive model, which enables a control-flow path to be abstracted by three contexts for its allocation, deallocation and dereference. This provides a balanced trade-off between precision and scalability.
- We present a static whole-program analysis that solves this three-point verification problem in the presence of both data-dependence and control-flow constraints. To this end, we develop a demand-driven pointer analysis with full context-sensitivity (modulo recursion) that automatically infers the context information required.
- We present an adaptive context-sensitive policy for TH-safety verification that automatically truncates redundant context prefixes without losing soundness or precision. This enables our approach to scale to some large real-world programs.

| | | |
|---|---|---|
| Program $P$ | ::= | $F^{+}$ |
| Function $F$ | ::= | ${}^{l}\texttt{def}\ f(\overrightarrow{x})\ \{\ S;\ \}$ |
| Statement $S$ | ::= | ${}^{l}x = y$ |
| | \| | ${}^{l}x = *y$ |
| | \| | ${}^{l}* x = y$ |
| | \| | ${}^{l}x = \&y$ |
| | \| | ${}^{l}x = \texttt{malloc()}$ |
| | \| | ${}^{l}\texttt{free}(y)$ |
| | \| | ${}^{l}x = fp(\overrightarrow{y})$ |
| | \| | ${}^{l}\texttt{ret}\ x$ |
| | \| | ${}^{l}\texttt{if}\ (*)\ S_1\ \texttt{else}\ S_2$ |
| | \| | ${}^{l}\texttt{while}\ (*)\ S$ |
| | \| | $S_1; S_2$ |

**Fig. 1.** A small unmanaged imperative language.

- We have implemented D$^3$ in LLVM and evaluated it using a suite of ten real-world programs. Our results show that D$^3$ scales to hundreds of KLOC, with a capability of verifying 81.3% of all the 114,508 dereferences to be safe.

## 2 Preliminaries

We describe our techniques using a small language in Figure 1. Function definitions and statements are identified by their labels or line numbers. The language is standard. Pointers are propagated via copy ($x = y$), load ($x = *y$), store ($*x = y$) and address-taking ($x = \&y$) statements; heap objects are allocated and deallocated by `malloc()` and `free()`, respectively; the callee of a function call ($x = fp(\overrightarrow{y})$) is specified by a function pointer ($fp$) with its parameters ($\overrightarrow{y}$) passed by value (as in LLVM-IR); and `ret`, `if` and `while` represent standard return, branching and looping statements.

As with previous work [8,58,36,34], we currently do not handle concurrent programs.

**Inter-Procedural Control-Flow Graph (ICFG)** This is a directed graph $(N, E)$, where each node $n \in N$ represents a statement and each edge $e = (src, dst) \in E$ represents the control flow from statement $src$ to statement $dst$. In particular, if $e$ represents a function call/return, then $e$ is labeled with the corresponding call-site ID $\kappa$.

**Contexts** Given any statement in function $f$, a *calling context* (or *context*, for short) $c = [\kappa_1, \kappa_2, ..., \kappa_n]$ is a sequence of $n$ call-site IDs in their invocation order that uniquely specifies an abstract call-path to $f$ on the ICFG of the program.

**Allocation, Deallocation and Dereference** A context-sensitive (abstract) *object*, denoted $\langle\langle c_o, o\rangle\rangle$, represents the set of concrete objects created at allocation site $o$ under

$$[\text{Addr}]\frac{\mathcal{P}[\![x = \&y]\!]}{y \in pt(x)} \qquad [\text{Alloc}]\frac{\mathcal{P}[\![^{o}x = \texttt{malloc()}]\!]}{o \in pt(x)}$$

$$[\text{Copy}]\frac{\mathcal{P}[\![x = y]\!]}{pt(y) \subseteq pt(x)}$$

$$[\text{Load}]\frac{\mathcal{P}[\![x = *y]\!] \qquad o \in pt(y)}{pt(o) \subseteq pt(x)} \qquad [\text{Store}]\frac{\mathcal{P}[\![*x = y]\!] \qquad o \in pt(x)}{pt(y) \subseteq pt(o)}$$

**Fig. 2.** Andersen-style subset-based, flow- and context-insensitive pointer analysis [4]. Passing arguments into and returning results from functions are handled as copy statements.

context $c_o$. We write $\psi(c_\psi, l_\psi, p)$ to signify a context-sensitive *deallocation* of the object pointed to by $p$ at line $l_\psi$ under context $c_\psi$. Similarly, a context-sensitive *dereference* $\omega(c_\omega, l_\omega, q)$ accesses the object pointed to by $q$ at line $l_\omega$ under context $c_\omega$. Context-insensitively, these notations are simplified to $o$, $\psi(l_\psi, p)$ and $\omega(l_\omega, q)$, respectively.

**Pointer Analysis** A context-sensitive pointer analysis conservatively computes a function $pt_{cs} : \mathrm{C} \times \mathrm{V} \to 2^{\mathrm{C}\times\mathrm{O}}$ that relates each context-sensitive pointer $(c, v) \in \mathrm{C} \times \mathrm{V}$ to the set of context-sensitive objects $\langle\!\langle c_o, o\rangle\!\rangle \in \mathrm{C}\times\mathrm{O}$ pointed to by $(c, v)$. A pointer analysis is formulated by a set of inference rules that can be solved using a standard fixed-point algorithm. Andersen-style [4] subset-based context-insensitive pointer analysis $pt : \mathrm{V} \to 2^{\mathrm{O}}$ is given in Figure 2. $\mathcal{P}[\![s]\!]$ signifies that statement $s$ appears in program $\mathcal{P}$.

We consider only field-sensitive pointer analysis techniques. As with previous techniques [6,22,39,58], we assume that our programs are ANSI-compliant that are devoid of buffer overflows and data misalignments. Arrays are handled monolithically. Any access to a member of an array or struct object with a statically unknown offset is viewed to be a non-deterministic operation on the given object (soundly but imprecisely).

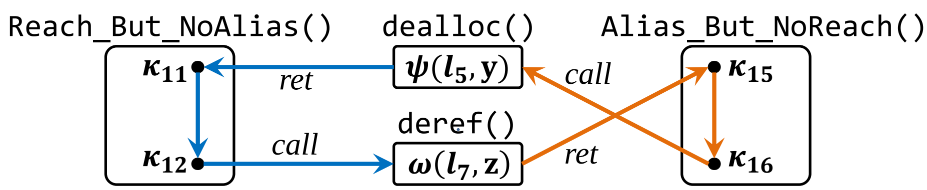
**TH-Safety Violation** A context-sensitive *TH-safety violation*, denoted $\langle\langle\!\langle c_o, o\rangle\!\rangle, \psi(c_\psi, l_\psi, p), \omega(c_\omega, l_\omega, q)\rangle$, occurs when $\langle\!\langle c_o, o\rangle\!\rangle$, which is deallocated at $l_\psi$ under $c_\psi$, is accessed later at $l_\omega$ under $c_\omega$. Our context-insensitive notation is $\langle o, \psi(l_\psi, p), \omega(l_\omega, q)\rangle$.

## 3 Illustrating Examples

In Section 3.1, we explain why aliasing and control-flow reachability must be solved synergistically rather than separately in order to achieve high precision in our verification task, no matter how precise pointer analysis is. In Section 3.2, we describe how our synergistic approach works on top of a demand-driven pointer analysis, by taming path explosion with full context-sensitivity (modulo recursion) adaptively.

```
1: def alloc() {
2:    x=malloc(); //o2
3:    ret x; }

4: def dealloc(y) {
5:    free(y); } //ψ(l5,y)

6: def deref(z) {
7:    temp=*z; } //ω(l7,z)

8: def Reach_But_NoAlias() {
9:    a=alloc();     //κ9 , ⟪[κ9 ],o2⟫
10:   b=alloc();     //κ10, ⟪[κ10],o2⟫
11:   dealloc(a);    //κ11
12:   deref(b); }    //κ12
13: def Alias_But_NoReach() {
14:   c=alloc();     //κ14, ⟪[κ14],o2⟫
15:   deref(c);      //κ15
16:   dealloc(c); }  //κ16
```

(a) Safe code, with a dereference in line 7




(b) ICFG (with relevant edges given), showing that $\psi(l_5, \texttt{y})$ reaches $\omega(l_7, \texttt{z})$ on the blue path but $\psi(l_5, \texttt{y})$ aliases with $\omega(l_7,\texttt{z})$ on the orange path, implying that the dereference at $l_7$ is safe

**Fig. 3.** An example without any TH-safety violation.

### 3.1 Aliasing and Control-Flow Reachability: Separately vs. Synergistically

Figure 3(a) gives a program, in which $\psi(l_5, y)$ does not cause a TH-safety violation at $\omega(l_7, z)$ (Figure 3(b)). The wrappers, `alloc()`, `dealloc()` and `deref()`, allocate $o_2$, deallocate the object pointed by `y` at $\psi(l_5, \texttt{y})$ and dereference `z` at $\omega(l_7, \texttt{z})$, respectively. In `Reach_But_NoAlias()`, $\langle\!\langle [\kappa_9], o_2 \rangle\!\rangle$ is first deallocated in $l_{11}$ and then another object $\langle\!\langle [\kappa_{10}], o_2 \rangle\!\rangle$ is accessed indirectly in $l_{12}$. In `Alias_But_NoReach()`, $\langle\!\langle [\kappa_{14}], o_2 \rangle\!\rangle$ is first accessed indirectly in $l_{15}$ and then deallocated in $l_{16}$.

If aliasing and control-flow reachability for $\psi(l_5, y)$ and $\omega(l_7, z)$ are solved separately, a TH-safety violation will be reported (but as a false positive), no matter how precise the underlying pointer analysis is used. As illustrated in Figure 3(b), aliasing (the orange path) and reachability (the blue path) happen along two different paths in the ICFG, and consequently, cannot be satisfied simultaneously in the same path.

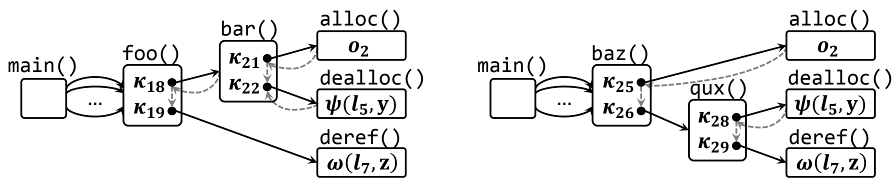
To avoid false positives like this, aliasing and control-flow reachability must be solved together. In our synergistic approach, we identify $o_2$, $\psi(l_5, \texttt{y})$ and $\omega(l_7, \texttt{z})$ by their respective contexts $c_o$, $c_\psi$ and $c_\omega$, and disprove the presence of a context tuple $(c_o, c_\psi, c_\omega)$, such that $\langle\!\langle c_o, o_2 \rangle\!\rangle$ is first deallocated in $l_5$ under $c_\psi$ and subsequently accessed in $l_7$ under $c_\omega$ along the same path. Therefore, our approach will report no TH-safety violation for this program. Note that any context-insensitive analysis that merges $\langle\!\langle [\kappa_9], o_2 \rangle\!\rangle$ and $\langle\!\langle [\kappa_{10}], o_2 \rangle\!\rangle$ into $o_2$ (by disregarding their contexts) will report a false violation as $\langle o_2, \psi(l_5, \texttt{y}), \omega(l_7, \texttt{z}) \rangle$.

```
17: def foo() {
18:    d=bar();    //κ18
19:    deref(d); }//κ19

20: def bar() {
21:    e=alloc(); //κ21, ⟪[κ21],o2⟫
22:    dealloc(e);//κ22
23:    ret e; }
```

(a) Allocation and deallocation (via wrappers) in the same function `bar()`

```
24: def baz() {
25:    f=alloc(); //κ25, ⟪[κ25],o2⟫
26:    qux(f); }  //κ26

27: def qux(g) {
28:    dealloc(g);//κ28
29:    deref(g); }//κ29
```

(b) Deallocation and dereference (via wrappers) in the same function `qux()`




(c) ICFG for (a)

(d) ICFG for (b)

**Fig. 4.** Two representative TH-safety violations caused by $\psi(l_5, y)$ and $\omega(l_7, z)$ appearing in Figure 3, where the three wrappers, `alloc()`, `dealloc()` and `deref()` are defined.

### 3.2 Synergizing Pointer Analysis and Control-Flow Reachability Analysis: On-Demand with Adaptive Context-Sensitivity

Let us illustrate our approach further by expanding Figure 3 into Figure 4 by examining how it detects two representative TH-safety violations caused now by $\psi(l_5, y)$ and $\omega(l_7, z)$ considered earlier. In Figure 4(a) (with its relevant ICFG given in Figure 4(c)), $o_2$ and $\psi(l_5, y)$ are reached transitively via the two call sites in the same function, `bar()`, which is called by `foo()`, in which $\omega(l_7, z)$ is reached via a call to `deref()` transitively. In Figure 4(b) (with its relevant ICFG given in Figure 4(d)), $\psi(l_5, y)$ and $\omega(l_7, z)$ are reached transitively via the two call sites in the same function, `qux()`, which is called by `baz()`, in which $o_2$ is reached via a call to `alloc()` transitively.

We will only discuss Figure 4(a) below as Figure 4(b) can be understood similarly.

***Verifying TH-Safety by Synergizing Pointer and Reachability Analyses On-Demand*** Our approach relies on $pt^{dd}_{cs}$, a *demand-driven* version of pointer analysis $pt_{cs}$ introduced in Section 2. For Figure 4(a), we report a TH-safety violation $\langle\!\langle\!\langle [\kappa_{18}, \kappa_{21}], o_2 \rangle\!\rangle, \psi([\kappa_{18}, \kappa_{22}], l_5, \texttt{y}), \omega([\kappa_{19}], l_7, \texttt{z})\rangle\!\rangle$. To obtain this, we check to see if `y` aliases `z` by querying $pt^{dd}_{cs}$ for the points-to sets of `y` and `z`, i.e., $pt^{dd}_{cs}([\,], \texttt{y})$ and $pt^{dd}_{cs}([\,], \texttt{z})$, respectively, where their initial unknown contexts [ ] will be eventually filled up by $pt^{dd}_{cs}$. On-demand, $pt^{dd}_{cs}$ traces backwards the flow of objects along the pre-computed def-use chains (obtained by a pre-analysis) in the program. To compute $pt^{dd}_{cs}([\,], \texttt{y})$, for example, starting from $l_5$, $pt^{dd}_{cs}$ traces back to $l_4$ where `y` is defined; moves to the call-site $\kappa_{22}$ where `y` receives the value of `e` via parameter passing; reaches $l_{21}$ where `e` is defined; encounters $l_3$ where `x` is returned (by entering `alloc()` from its exit at $\kappa_{21}$); and finally, arrives at $l_2$ where x is defined, giving rise to $\langle\!\langle[\kappa_{21}], o_2\rangle\!\rangle \in pt^{dd}_{cs}([\kappa_{22}], \texttt{y})$. Note that the initial unknown context [ ] has been inferred to be $[\kappa_{22}]$ as desired. This implies that $\langle\!\langle[\kappa_{18}, \kappa_{21}], o_2\rangle\!\rangle \in pt^{dd}_{cs}([\kappa_{18}, \kappa_{22}], \texttt{y})$. Similarly we obtain $\langle\!\langle[\kappa_{18}, \kappa_{21}], o_2\rangle\!\rangle \in pt^{dd}_{cs}([\kappa_{19}], \texttt{z})$. Thus, $\psi([\kappa_{18}, \kappa_{22}], l_5, \texttt{y})$ aliases with $\omega([\kappa_{19}], l_7, \texttt{z})$ (with y and z both pointing to $\langle\!\langle[\kappa_{18}, \kappa_{21}], o_2\rangle\!\rangle$), and in addition, the former also reaches the latter along the same path identified by $[\kappa_{18}, \kappa_{21}], [\kappa_{18}, \kappa_{22}]$ and $[\kappa_{19}]$. As a result, our approach reports this violation as $\langle\langle\!\langle[\kappa_{18}, \kappa_{21}], o_2\rangle\!\rangle, \psi([\kappa_{18}, \kappa_{22}], l_5, \texttt{y}), \omega([\kappa_{19}], l_7, \texttt{z})\rangle$.

***Taming Path Explosion with Adaptive Context-Sensitivity*** In our approach, $pt^{dd}_{cs}$ applies context-sensitivity adaptively without analyzing the callers of `foo()`, avoiding the possible path explosion that may occur between `main()` and `foo()` in Figure 4(c). Soundness is still guaranteed, since the context elements between `main()` and `foo()` do not affect the value-flows of $\langle\!\langle[\kappa_{18}, \kappa_{21}], o_2\rangle\!\rangle$ and are thus redundant. To see this, if we extend the two contexts in $\psi([\kappa_{18}, \kappa_{22}], l_5, \texttt{y})$ and $\omega([\kappa_{19}], l_7, \texttt{z})$ with two distinct prefixes, $[\kappa_{a1}]$ and $[\kappa_{a2}]$, we will fail to obtain any additional violation witness, since both are no longer aliased: $pt^{dd}_{cs}([\kappa_{a1}, \kappa_{18}, \kappa_{22}], \texttt{y}) = \{\langle\!\langle[\kappa_{a1}, \kappa_{18}, \kappa_{21}], o_2\rangle\!\rangle\} \neq \{\langle\!\langle[\kappa_{a2}, \kappa_{18}, \kappa_{21}], o_2\rangle\!\rangle\} = pt^{dd}_{cs}([\kappa_{a2}, \kappa_{19}], \texttt{z})$. If we use the the same prefix instead, we will end up with a finer abstraction, yielding the results already subsumed.

# 4 Our Approach

The workflow of our four-stage approach is given in Figure 5. To start with (①), we perform a fast but imprecise pre-analysis for a program using Andersen's pointer analysis $pt$ (Figure 2). Then (②), we build a value-flow graph to capture the flow of values across the program based on the points-to information obtained in the pre-analysis (Section 4.1). Next (③), we obtain the points-to set at each dereference by querying $pt^{dd}_{cs}$, a demand-driven version of $pt_{cs}$ (discussed in Section 2) that now operates on the value-flow graph (Section 4.2). This way, $pt^{dd}_{cs}$ will traverse pre-computed def-use chains rather than control-flow, achieving better efficiency. Finally (④), we verify absence of a TH-safety violation at a dereference by considering aliasing and control-flow reachability synergistically with adaptive context-sensitivity (Sections 4.3 and 4.4).

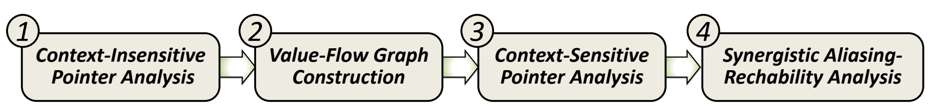



**Fig. 5.** The workflow of our approach on synergizing pointer analysis with reachability analysis.

## 4.1 Value-Flow Graph Construction

We construct a *value-flow graph* for a program, following [12,49,44], based on the points-to information discovered during the pre-analysis to capture the flow of values across the program. This entails building the def-use chains for its top-level variables

$$[\text{Mu}]\ \frac{\mathcal{P}[\![^{l}\ x = *y]\!] \qquad o \in pt(y)}{[\![\mu(o)]\!] \in \Delta(l, \prec)} \qquad\qquad [\text{Chi}]\ \frac{\mathcal{P}[\![^{l}\ *x = y]\!] \qquad o \in pt(x)}{[\![o = \chi(o)]\!] \in \Delta(l, \succ)}$$

$$[\text{Ref}]\ \frac{\begin{array}{cc} \mathcal{P}[\![^{l}\ \_ = fp(\_)]\!] & f \in pt(fp) \\ \mathcal{P}[\![^{l_f}\ \texttt{def}\ f(\_)\{...\}]\!] & l_s \in L(f) \\ \multicolumn{2}{c}{[\![\mu(o)]\!] \in \Delta(l_s, \prec)} \end{array}}{[\![\mu(o)]\!] \in \Delta(l, \prec) \quad [\![o = \chi(o)]\!] \in \Delta(l_f, \prec)} \qquad [\text{Mod}]\ \frac{\begin{array}{cc} \mathcal{P}[\![^{l}\ \_ = fp(\_)]\!] & f \in pt(fp) \\ \mathcal{P}[\![^{l_f}\ \texttt{def}\ f(\_)\{...\}]\!] & l_s \in L(f) \\ \multicolumn{2}{c}{[\![o = \chi(o)]\!] \in \Delta(l_s, \succ)} \end{array}}{\begin{array}{cc} [\![\mu(o)]\!] \in \Delta(l, \prec) & [\![o = \chi(o)]\!] \in \Delta(l_f, \prec) \\ [\![\mu(o)]\!] \in \Delta(l_f, \succ) & [\![o = \chi(o)]\!] \in \Delta(l, \succ) \end{array}}$$

**Fig. 6.** Rules for adding two types of annotations, $[\![\mu(o)]\!]$ and $[\![o = \chi(o)]\!]$, to make explicit the accesses of a memory object $o$. $L(f)$ denotes the set of statement labels in function $f$. $\Delta(l, \prec)$ and $\Delta(l, \succ)$ represent the sets of annotations added just before and after $l$, respectively.

(which are conceptually regarded as register variables) and address-taken variables (which are all referred to as memory objects or objects for short in this paper).

The def-use chains for top-level variables are readily available. However, those for address-taken variables (accessed indirectly at loads, stores and call sites) are implicit. To make such indirect memory accesses explicit, we resort to the rules in Figure 6. For an address-taken variable $o$, there are two types of annotations: $[\![\mu(o)]\!]$, which represents a potential use of $o$, and $[\![o = \chi(o)]\!]$, which represents both a potential definition and a potential use of $o$. We define $\Delta : \mathrm{L} \times \mathrm{ORD} \to 2^{\mathrm{ANNOT}}$, where ANNOT is the set of annotations (shown in brackets), L is the set of statement labels, and $\mathrm{ORD} = \{\prec, \succ\}$ indicates if an annotation appears immediately before ($\prec$) or after ($\succ$) a statement.

Let us go through the rules in Figure 6, where allow us to soundly model both strong updates (by killing old values) and weak updates (by preserving old values) for address-taken variables. For a load statement $x = *y$ at $l$, if $y$ points to $o$, then $[\![\mu(o)]\!]$ is added before $l$ to indicate that $o$ may be used at this load (Rule [Mu]). For a store statement $*x = y$ at $l$, if $x$ points to $o$, then $[\![o = \chi(o)]\!]$ is added after $l$ to indicate that $o$ (LHS) may be redefined in terms of both $o$ (RHS) in the case of a weak update and $y$ at this store (Rule [Chi]). Rules [Ref] and [Mod] prescribe the standard inter-procedural MOD/REF analysis. Let a function $f$ be defined at $l_f$ and called at a call site $l$ via a function pointer $fp$. Consider [Ref] first. If $[\![\mu(o)]\!]$ is annotated inside $f$, then $[\![\mu(o)]\!]$ is added before $l$ (as $o$ may be used in $f$ directly or indirectly), and $[\![o = \chi(o)]\!]$ is added before $f$'s definition at $l_f$ (as $o$ may be passed indirectly as a parameter to $f$). Consider [Mod] now. If $[\![o = \chi(o)]\!]$ is annotated inside $f$, then we add not only the same annotations at $l$ and $l_f$ as in [Ref], but also $[\![\mu(o)]\!]$ after $l_f$ (as $o$ may be returned to its call sites) and $[\![o = \chi(o)]\!]$ after $l$ (as $o$ may be modified at $l$).

Once a program has been annotated, its top-level variables and objects appearing in the annotations are put into SSA form [11], with their versions denoted in *superscripts*.

**Example 1.** Let us see how to add $o_5$-related annotations in Figure 7. For now, the value-flow edges shown are irrelevant. In line 8, $[\![o_5^2 = \chi(o_5^1)]\!]$ is added after $l_8$, i.e., as $8^{\succ}$, in `put()` as `pctn` is found to point to $o_5$ by the pre-analysis in Figure 2 (Rule [Chi]). As a result, this inter-procedural MOD/REF effect needs to be reflected at its

```
1:  def mkFd() {
2:     fd=malloc();    //o_2
3:     ret fd;
    }

4:  def mkCtn() {
5:     ctn=malloc();   //o_5
6:     ret ctn;
    }

7<: [[o_5^1 = χ(o_5^0)]]
7:  def put(pfd, pctn) {
8:     *pctn=pfd;
8>:    [[o_5^2 = χ(o_5^1)]]
    }
7>: [[μ(o_5^2)]]

9:  def feedPets() {
10:    bone=mkFd();  //<<[κ_10],o_2>>
11:    fish=mkFd();  //<<[κ_11],o_2>>
12:    tray=mkCtn(); //<<[κ_12],o_5>>
13:    bowl=mkCtn(); //<<[κ_13],o_5>>

14<:   [[μ(o_5^0)]]
14:    put(bone, tray);//κ_14
14>:   [[o_5^1 = χ(o_5^0)]]

15<:   [[μ(o_5^1)]]
15:    put(fish, bowl);//κ_15
15>:   [[o_5^2 = χ(o_5^1)]]

16<:   [[μ(o_5^2)]]
16:    feedDog=*tray;
    }
```

**Fig. 7.** A program (referred to in Example 1 (annotations), Example 2 (value-flow edges) and Example 3 (pointer analysis)), decorated with $\mu$ and $\chi$ annotations and all the value-flow edges 1 – 9 that capture the flow of $o_2$ from `bone` in line 10 through to `feedDog` in line 16.

definition and call sites, by adding $7^{\lessdot}$, $7^{\gtrdot}$, $14^{\lessdot}$, $14^{\gtrdot}$, $15^{\lessdot}$, and $15^{\gtrdot}$ (Rule [Mod]). In line 16, $[\![\mu(o_5^2)]\!]$ is added before $l_{16}$ since `tray` is found to point to $o_5$ (Rule [Mu]).

Given an annotated program in SSA form, we build its value-flow graph, $G_{\text{vfg}} = (\mathrm{L} \times \mathrm{V}, \mathrm{E})$, to capture the flow of values through its def-use chains and inter-procedural call/return edges, by using the rules in Figure 8 to construct its value-flow edges. We make use of two mappings, $\mathcal{D} : \mathrm{V} \to 2^{\mathrm{L}}$ and $\mathcal{U} : \mathrm{V} \to 2^{\mathrm{L}}$, that map a variable $v \in \mathrm{V}$ to the set of its definition sites $l_{def} \in L$ and use sites $l_{use} \in L$, respectively. We write $\langle l_{src}, v\rangle \longrightarrow \langle l_{dst}, v'\rangle$ to denote the flow of a value initially in $v$ at $l_{src}$ to $v'$ at $l_{dst}$. For a top-level variable $x \in \mathrm{V}^{\mathrm{T}}$, Rule [$\mathrm{D}^{\mathrm{T}}$] adds the definition site $l$ to $\mathcal{D}(x)$ and Rules [$\mathrm{U}^{\mathrm{T}}_{\mathrm{Copy}}$], [$\mathrm{U}^{\mathrm{T}}_{\mathrm{Load}}$], [$\mathrm{U}^{\mathrm{T}}_{\mathrm{Store}}$], [$\mathrm{U}^{\mathrm{T}}_{\mathrm{Addr}}$], [$\mathrm{U}^{\mathrm{T}}_{\mathrm{Free}}$] and [$\mathrm{U}^{\mathrm{T}}_{\mathrm{Call}}$] add the use site $l$ to $\mathcal{U}(x)$. For an address-taken variable $o \in \mathrm{V}^{\mathrm{A}}$, Rules [$\mathrm{D}^{\mathrm{A}}$] and [$\mathrm{U}^{\mathrm{A}}_{\chi}$]/[$\mathrm{U}^{\mathrm{A}}_{\mu}$] simply collect its definition and use sites into $\mathcal{D}(o)$ and $\mathcal{U}(o)$, respectively. The last five rules construct the edges in $G_{\text{vfg}}$ by connecting a definition site with all its use sites. [$\mathrm{VF}^{\mathrm{Intra}}$] adds intra-procedural value-flow edges while the other four add inter-procedural value-flow edges (with [$\mathrm{VF}^{\mathrm{T}}_{\mathrm{Call}}$] and [$\mathrm{VF}^{\mathrm{T}}_{\mathrm{Ret}}$] for top-level variables and [$\mathrm{VF}^{\mathrm{A}}_{\mathrm{Call}}$] and [$\mathrm{VF}^{\mathrm{A}}_{\mathrm{Ret}}$] for address-taken variables).

Once $G_{\text{vfg}}$ has been constructed, the SSA versions of a variable will be ignored.

**Example 2.** Figure 7 shows all the value-flow edges 1 – 9 capturing the flow of $o_2$ via `fd`, `bone`, `pfd`, $o_5$ and `feedDog`. We obtain these edges by applying the following rules (Figure 8): 1 for $\langle l_2, \texttt{fd}\rangle \longrightarrow \langle l_3, \texttt{fd}\rangle$ ([$\mathrm{VF}^{\mathrm{Intra}}$]); 2 for $\langle l_3, \texttt{fd}\rangle \longrightarrow \langle l_{10}, \texttt{bone}\rangle$ ([$\mathrm{VF}^{\mathrm{T}}_{\mathrm{Ret}}$]); 3 for $\langle l_{10}, \texttt{bone}\rangle \longrightarrow \langle l_{14}, \texttt{bone}\rangle$ ([$\mathrm{VF}^{\mathrm{Intra}}$]); 4 for $\langle l_{14}, \texttt{bone}\rangle \longrightarrow \langle l_7, \texttt{pfd}\rangle$ ([$\mathrm{VF}^{\mathrm{T}}_{\mathrm{Call}}$]); 5 for $\langle l_7, \texttt{pfd}\rangle \longrightarrow \langle l_8, \texttt{pfd}\rangle$ and 6 for $\langle l_8^{\gtrdot}, o_5^2\rangle \longrightarrow \langle l_7^{\gtrdot}, o_5^2\rangle$ ([$\mathrm{VF}^{\mathrm{Intra}}$]); 7 for $\langle l_7^{\gtrdot}, o_5^2\rangle \longrightarrow \langle l_{14}^{\gtrdot}, o_5^0\rangle$ ([$\mathrm{VF}^{\mathrm{A}}_{\mathrm{Ret}}$]); and 8 for $\langle l_{14}^{\gtrdot}, o_5^1\rangle \longrightarrow \langle l_{15}^{\gtrdot}, o_5^1\rangle$ and 9 for $\langle l_{15}^{\gtrdot}, o_5^2\rangle \longrightarrow \langle l_{16}^{\lessdot}, o_5^2\rangle$ ([$\mathrm{VF}^{\mathrm{Intra}}$]).

$$[\mathrm{D}^{\mathrm{T}}]\,\frac{\mathcal{P}[\![^{l}\; x = \_]\!]}{l \in \mathcal{D}(x)} \qquad [\mathrm{U}^{\mathrm{T}}_{\mathrm{Copy}}]\,\frac{\mathcal{P}[\![^{l}\; \_ = x]\!]}{l \in \mathcal{U}(x)} \qquad [\mathrm{U}^{\mathrm{T}}_{\mathrm{Load}}]\,\frac{\mathcal{P}[\![^{l}\; \_ = *x]\!]}{l \in \mathcal{U}(x)} \qquad [\mathrm{U}^{\mathrm{T}}_{\mathrm{Store}}]\,\frac{\mathcal{P}[\![^{l}\; *x = \_]\!]}{l \in \mathcal{U}(x)}$$

$$[\mathrm{U}^{\mathrm{T}}_{\mathrm{Addr}}]\,\frac{\mathcal{P}[\![^{l}\; \_ = \&x]\!]}{l \in \mathcal{U}(x)} \qquad [\mathrm{U}^{\mathrm{T}}_{\mathrm{Free}}]\,\frac{\mathcal{P}[\![^{l}\, \mathtt{free}(x)]\!]}{l \in \mathcal{U}(x)} \qquad [\mathrm{U}^{\mathrm{T}}_{\mathrm{Call}}]\,\frac{\mathcal{P}[\![^{l}\; \_ = fp(\vec{x})]\!] \quad x \in \vec{x}}{l \in \mathcal{U}(x)}$$

$$[\mathrm{D}^{\mathrm{A}}]\,\frac{[\![o = \_]\!] \in \Delta(l, \succ)}{l^{\succ} \in \mathcal{D}(o)} \qquad [\mathrm{U}^{\mathrm{A}}_{\chi}]\,\frac{[\![\_ = \chi(o)]\!] \in \Delta(l, \succ)}{l^{\succ} \in \mathcal{U}(o)} \qquad [\mathrm{U}^{\mathrm{A}}_{\mu}]\,\frac{[\![\mu(o)]\!] \in \Delta(l, \prec)}{l^{\prec} \in \mathcal{U}(o)}$$

$$[\mathrm{VF}^{\mathrm{Intra}}]\,\frac{l_d \in \mathcal{D}(o) \quad l_u \in \mathcal{U}(o) \quad F(l_d) = F(l_u)}{\langle l_d, o\rangle \longrightarrow \langle l_u, o\rangle}$$

$$[\mathrm{VF}^{\mathrm{T}}_{\mathrm{Call}}]\,\frac{\begin{array}{c}\mathcal{P}[\![^{l}\; \_ = fp(\vec{x})]\!] \quad x_i \in \vec{x} \quad y_i \in \vec{y} \\ \mathcal{P}[\![^{l_f}\, \mathtt{def}\ f(\vec{y})\{\ldots\}]\!] \quad f \in pt(fp)\end{array}}{\langle l, x_i\rangle \longrightarrow \langle l_f, y_i\rangle} \qquad [\mathrm{VF}^{\mathrm{T}}_{\mathrm{Ret}}]\,\frac{\begin{array}{c}\mathcal{P}[\![^{l}\; y = fp(\_)]\!] \quad f \in pt(fp) \\ \mathcal{P}[\![^{l_f}\, \mathtt{def}\ f(\_)\{\ldots\ ^{l_r}\ \mathtt{ret}\ x;\}]\!]\end{array}}{\langle l_r, x\rangle \longrightarrow \langle l, y\rangle}$$

$$[\mathrm{VF}^{\mathrm{A}}_{\mathrm{Call}}]\,\frac{\begin{array}{c}\mathcal{P}[\![^{l}\; \_ = fp(\_)]\!] \quad [\![\mu(o^{i})]\!] \in \Delta(l, \prec) \\ \mathcal{P}[\![^{l_f}\, \mathtt{def}\ f(\_)\{\ldots\}]\!] \quad f \in pt(fp) \\ [\![\_ = \chi(o^{j})]\!] \in \Delta(l_f, \prec)\end{array}}{\langle l^{\prec}, o^{i}\rangle \longrightarrow \langle l_f^{\prec}, o^{j}\rangle} \qquad [\mathrm{VF}^{\mathrm{A}}_{\mathrm{Ret}}]\,\frac{\begin{array}{c}\mathcal{P}[\![^{l}\; \_ = fp(\_)]\!] \quad [\![\_ = \chi(o^{j})]\!] \in \Delta(l, \succ) \\ \mathcal{P}[\![^{l_f}\, \mathtt{def}\ f(\_)\{\ldots\}]\!] \quad f \in pt(fp) \\ [\![\mu(o^{i})]\!] \in \Delta(l_f, \succ)\end{array}}{\langle l_f^{\succ}, o^{i}\rangle \longrightarrow \langle l^{\succ}, o^{j}\rangle}$$

**Fig. 8.** Rules for building the value-flow graph $G_{\mathrm{vfg}}$ for an annotated program in SSA form (with the version of an SSA variable omitted when it is irrelevant to avoid cluttering). $\mathcal{D}(v)$ ($\mathcal{U}(v)$) denotes the set of definition (use) sites of a variable $v$. $F(l)$ identifies the function containing $l$.

In Figure 10 (discussed in Section 4.2), we will give a version of Figure 7 with all the value-flow edges included for the program.

### 4.2 Demand-Driven Context-Sensitive Pointer Analysis

Our context-sensitive pointer analysis $pt^{dd}_{cs}$ operates on the value-flow graph $G_{\mathrm{vfg}}$ of a program. We write $pt^{dd}_{cs}(c, l, v) = \diamondplus$ to signify a demand query for the points-to set of variable $v$ at statement $l$ under context $c$. In the case of $pt^{dd}_{cs}([\ ], l, v) = \diamondplus$ with an empty context $[\ ]$, $pt^{dd}_{cs}$ will find all pointed-to objects $\langle\!\langle c_o, o\rangle\!\rangle \in pt^{dd}_{cs}(c, l, v)$, where $c$ is also inferred automatically. This automatic context inference is essential for achieving high precision as it provides a mechanism for us to synergize alias and control-flow reachability analyses as needed. As $pt^{dd}_{cs}(c, l, v) = \diamondplus$ is solved on-demand (with possibly many other points-to queries raised along the way), by traversing backwards only the value-flow edges in $G_{\mathrm{vfg}}$ established on the fly, imprecision in $G_{\mathrm{vfg}}$ (due to spurious value-flow edges) will affect only the efficiency but not precision of $pt^{dd}_{cs}$.

$$[\text{QRY}]\,\frac{pt^{dd}_{cs}(c,l,v) = \Diamond}{\Diamond \rightsquigarrow \langle c,l,v\rangle} \qquad [\text{PT}]\,\frac{\langle\!\langle c_o,o\rangle\!\rangle \rightsquigarrow \langle c,l,v\rangle}{\langle\!\langle c_o,o\rangle\!\rangle \in pt^{dd}_{cs}(c,l,v)} \qquad [\text{DD}_{\text{Back}}]\,\frac{\langle c,l,v\rangle \rightsquigarrow \_}{\Diamond \rightsquigarrow \langle c,l,v\rangle}$$

$$[\text{VF}_{\text{Addr}}]\,\frac{\mathcal{P}\llbracket^{l}\ x = \&y\rrbracket \quad \Diamond \rightsquigarrow \langle c,l,x\rangle}{\langle\!\langle c,y\rangle\!\rangle \rightsquigarrow \langle c,l,x\rangle} \qquad [\text{VF}_{\text{Alloc}}]\,\frac{\mathcal{P}\llbracket^{o}\ x = \texttt{malloc()}\rrbracket \quad \Diamond \rightsquigarrow \langle c,l,x\rangle}{\langle\!\langle c,o\rangle\!\rangle \rightsquigarrow \langle c,l,x\rangle}$$

$$[\text{DD}_{\text{Load}}]\,\frac{\mathcal{P}\llbracket^{l}\ x = *y\rrbracket \quad \Diamond \rightsquigarrow \langle c,l,x\rangle}{\Diamond \rightsquigarrow \langle c,l,y\rangle} \qquad [\text{VF}_{\text{Load}}]\,\frac{\begin{array}{c}\mathcal{P}\llbracket^{l}\ x = *y\rrbracket \quad \Diamond \rightsquigarrow \langle c,l,x\rangle\\ \langle\!\langle c_o,o\rangle\!\rangle \rightsquigarrow \langle c,l,y\rangle\end{array}}{\langle c,l^{\prec},\langle\!\langle c_o,o\rangle\!\rangle\rangle \rightsquigarrow \langle c,l,x\rangle}$$

$$[\text{DD}_{\text{Store}}]\,\frac{\mathcal{P}\llbracket^{l}\ *x = y\rrbracket \quad \Diamond \rightsquigarrow \langle c,l^{\succ},\langle\!\langle c_o,o\rangle\!\rangle\rangle}{\Diamond \rightsquigarrow \langle c,l,x\rangle} \qquad [\text{VF}_{\text{Store}}]\,\frac{\begin{array}{c}\mathcal{P}\llbracket^{l}\ *x = y\rrbracket \quad \Diamond \rightsquigarrow \langle c,l^{\succ},\langle\!\langle c_o,o\rangle\!\rangle\rangle\\ \langle\!\langle c_o,o\rangle\!\rangle \rightsquigarrow \langle c,l,x\rangle\end{array}}{\langle c,l,y\rangle \rightsquigarrow \langle c,l^{\succ},\langle\!\langle c_o,o\rangle\!\rangle\rangle}$$

$$[\text{VF}_{\text{Copy}}]\,\frac{\mathcal{P}\llbracket^{l}\ x = y\rrbracket \quad \Diamond \rightsquigarrow \langle c,l,x\rangle}{\langle c,l,y\rangle \rightsquigarrow \langle c,l,x\rangle} \qquad [\text{VF}_{\text{Trans}}]\,\frac{\begin{array}{c}\langle c',l',v'\rangle \rightsquigarrow \langle c'',l'',v''\rangle\\ \langle c'',l'',v''\rangle \rightsquigarrow \langle c''',l''',v'''\rangle\end{array}}{\langle c',l',v'\rangle \rightsquigarrow \langle c''',l''',v'''\rangle}$$

$$[\text{VF}^{\text{T}}]\,\frac{\begin{array}{c}\langle l',x\rangle \longrightarrow \langle l,x\rangle \qquad x \in \text{V}^{\text{T}}\\ \Diamond \rightsquigarrow \langle c,l,x\rangle\end{array}}{\langle c,l',x\rangle \rightsquigarrow \langle c,l,x\rangle} \qquad [\text{VF}^{\text{A}}]\,\frac{\begin{array}{c}\langle l',o\rangle \longrightarrow \langle l,o\rangle \qquad o \in \text{V}^{\text{A}}\\ \Diamond \rightsquigarrow \langle c,l,\langle\!\langle c_o,o\rangle\!\rangle\rangle\end{array}}{\langle c,l',\langle\!\langle c_o,o\rangle\!\rangle\rangle \rightsquigarrow \langle c,l,\langle\!\langle c_o,o\rangle\!\rangle\rangle}$$

$$[\text{VF}^{\text{T}}_{\text{Call}}]\,\frac{\begin{array}{c}\mathcal{P}\llbracket^{l}\ \_ = fp(\vec{x})\rrbracket \quad \mathcal{P}\llbracket^{l_f}\,\texttt{def}\ f(\vec{y})\{...\}\rrbracket\\ \Diamond \rightsquigarrow \langle c,l_f,y\rangle \qquad c^{-} = c \ominus l\\ \langle l,x\rangle \longrightarrow \langle l_f,y\rangle\end{array}}{\langle c^{-},l,x\rangle \rightsquigarrow \langle c,l_f,y\rangle} \qquad [\text{VF}^{\text{T}}_{\text{Ret}}]\,\frac{\begin{array}{c}\mathcal{P}\llbracket^{l}\ x = fp(\_)\rrbracket \quad \mathcal{P}\llbracket^{l_r}\,\texttt{ret}\ y\rrbracket\\ \Diamond \rightsquigarrow \langle c,l,x\rangle \qquad c^{+} = c \oplus l\\ \langle l_r,y\rangle \longrightarrow \langle l,x\rangle\end{array}}{\langle c^{+},l_r,y\rangle \rightsquigarrow \langle c,l,x\rangle}$$

$$[\text{VF}^{\text{A}}_{\text{Call}}]\,\frac{\begin{array}{c}\mathcal{P}\llbracket^{l}\ \_ = fp(\_)\rrbracket \quad \mathcal{P}\llbracket^{l_f}\,\texttt{def}\ f(\_)\ \{...\}\rrbracket\\ \Diamond \rightsquigarrow \langle c,l_f^{\prec},\langle\!\langle c_o,o\rangle\!\rangle\rangle \qquad c^{-} = c \ominus l\\ \langle l,o\rangle \longrightarrow \langle l_f^{\prec},o\rangle\end{array}}{\langle c^{-},l,\langle\!\langle c_o,o\rangle\!\rangle\rangle \rightsquigarrow \langle c,l_f^{\prec},\langle\!\langle c_o,o\rangle\!\rangle\rangle} \qquad [\text{VF}^{\text{A}}_{\text{Ret}}]\,\frac{\begin{array}{c}\mathcal{P}\llbracket^{l}\ \_ = fp(\_)\rrbracket \quad \mathcal{P}\llbracket^{l_f}\,\texttt{def}\ f(\_)\ \{...\}\rrbracket\\ \Diamond \rightsquigarrow \langle c,l,\langle\!\langle c_o,o\rangle\!\rangle\rangle \qquad c^{+} = c \oplus l\\ \langle l_f^{\succ},o\rangle \longrightarrow \langle l,o\rangle\end{array}}{\langle c^{+},l_f^{\succ},\langle\!\langle c_o,o\rangle\!\rangle\rangle \rightsquigarrow \langle c,l,\langle\!\langle c_o,o\rangle\!\rangle\rangle}$$

**Fig. 9.** Rules for demand-driven context-sensitive pointer analysis $pt^{dd}_{cs}$ (with $\Diamond$ denoting a demand query issued and $n_{src} \rightsquigarrow n_{dst}$ denoting the flow of a value from $n_{src}$ to $n_{dst}$ on $G_{\text{vfg}}$).

Figure 9 gives the rules for answering $pt^{dd}_{cs}(c,l,v) = \Diamond$, where $\rightsquigarrow$, which is transitive by [VF$_{\text{Trans}}$], represents the flow of a value across one or more value-flow edges in $G_{\text{vfg}}$ actually traversed. Note that $\langle\!\langle c_o,o\rangle\!\rangle$ is essentially $\langle c_o,o,o\rangle$ since $o$ is the line number for the corresponding allocation site. We say that $x$ *flows to* $y$ if

$\langle \_, \_, x\rangle \rightsquigarrow \langle \_, \_, y\rangle$. To solve $pt^{dd}_{cs}(c, l, v) = \diamondsuit$, we solve $\diamondsuit \rightsquigarrow \langle c, l, v\rangle$, i.e., find what flows to $\langle c, l, v\rangle$ (Rule [QRY]). If $\langle\!\langle c_o, o\rangle\!\rangle$ flows to $\langle c, l, v\rangle$, then $\langle c, l, v\rangle$ points to $\langle\!\langle c_o, o\rangle\!\rangle$ (Rule [PT]). If $\langle c, l, v\rangle$ has been reached, we need to continue exploring backwards what may flow to $\langle c, l, v\rangle$ on-demand (Rule [$\text{DD}_{\text{Back}}$]). Rules [$\text{VF}_{\text{Addr}}$] and [$\text{VF}_{\text{Alloc}}$] handle allocation statements that allocate memory for an address-taken variable on the stack and in the heap, respectively.

For a load ${}^{l}x = *y$ with a query $\diamondsuit \rightsquigarrow \langle c, l, x\rangle$, $pt^{dd}_{cs}$ first checks to see if $\langle\!\langle c_o, o\rangle\!\rangle \rightsquigarrow \langle c, l, y\rangle$ holds by issuing a demand query $\diamondsuit \rightsquigarrow \langle c, l, y\rangle$ (Rule [$\text{DD}_{\text{Load}}$]), and if this is the case, then $\langle c, l^{<}, \langle\!\langle c_o, o\rangle\!\rangle\rangle \rightsquigarrow \langle c, l, x\rangle$ is established (Rule [$\text{VF}_{\text{Load}}$]). Similarly, for a store ${}^{l}*x = y$ with a query $\diamondsuit \rightsquigarrow \langle c, l^{>}, \langle\!\langle c_o, o\rangle\!\rangle\rangle$, $pt^{dd}_{cs}$ checks to see if $\langle\!\langle c_o, o\rangle\!\rangle \rightsquigarrow \langle c, l, x\rangle$ holds by issuing a demand query $\diamondsuit \rightsquigarrow \langle c, l, x\rangle$ (Rule [$\text{DD}_{\text{Store}}$]), and if this is the case, then $\langle c, l, y\rangle \rightsquigarrow \langle c, l^{>}, \langle\!\langle c_o, o\rangle\!\rangle\rangle$ is established (Rule [$\text{VF}_{\text{Store}}$]).

Rules [$\text{VF}_{\text{Copy}}$], [$\text{VF}^{\text{T}}$] and [$\text{VF}^{\text{A}}$] simply propagate values across assignments (with the former for copy statements and the latter two for def-use chains). In particular, [$\text{VF}^{\text{A}}$] performs a weak update at a store. Note that $pt^{dd}_{cs}$ is also flow-sensitive with strong updates performed for singleton objects as is standard [29,19,49].

To support the inter-procedural analysis at the function calls and returns, [$\text{VF}^{\text{T}}_{\text{Call}}$] and [$\text{VF}^{\text{T}}_{\text{Ret}}$] handle top-level variables while [$\text{VF}^{\text{A}}_{\text{Call}}$] and [$\text{VF}^{\text{A}}_{\text{Ret}}$] handle address-taken variables. Context-sensitivity is achieved by maintaining a context with push ($\oplus$) and pop ($\ominus$) operations in a stack-like manner. When handling a function call at a call site $l$, a new context $c^{-}$ is generated by popping off $l$ from the current context $c$, denoted $c^{-} = c \ominus l$, to track the value-flow backwards outside the callee ($c^{-}$) from inside the callee ($c$). Conversely, when handling a callee function's return statement that returns to a call site $l$, a new context $c^{+}$ is created by pushing $l$ to the top of the current context $c$, denoted $c^{+} = c \oplus l$, to represent the fact that the backward analysis will now enter the callee ($c^{+}$) at its return statement from the call-site $l$ outside the callee ($c$).

**Example 3.** Given $pt^{dd}_{cs}([\,], 16, \texttt{feedDog}) = \diamondsuit$ for the program in Figure 7, $pt^{dd}_{cs}$ yields the following facts related to the nine value-flow edges marked as 1 – 9:

$$
\boxed{\begin{array}{llll}
\quad \langle\!\langle [\kappa_{10}], o_2\rangle\!\rangle & \rightsquigarrow \langle [\kappa_{10}], 2, \texttt{fd}\rangle & \overset{1}{\rightsquigarrow} \langle [\kappa_{10}], 3, \texttt{fd}\rangle & \\
\overset{2}{\rightsquigarrow} \langle [\,], 10, \texttt{bone}\rangle & \overset{3}{\rightsquigarrow} \langle [\,], 14, \texttt{bone}\rangle & \overset{4}{\rightsquigarrow} \langle [\kappa_{14}], 7, \texttt{pfd}\rangle & \\
\overset{5}{\rightsquigarrow} \langle [\kappa_{14}], 8, \texttt{pfd}\rangle & \rightsquigarrow \langle [\kappa_{14}], 8^{>}, \langle\!\langle [\kappa_{12}], o_5\rangle\!\rangle\rangle & \overset{6}{\rightsquigarrow} \langle [\kappa_{14}], 7^{>}, \langle\!\langle [\kappa_{12}], o_5\rangle\!\rangle\rangle & \\
\overset{7}{\rightsquigarrow} \langle [\,], 14^{>}, \langle\!\langle [\kappa_{12}], o_5\rangle\!\rangle\rangle & \overset{8}{\rightsquigarrow} \langle [\,], 15^{>}, \langle\!\langle [\kappa_{12}], o_5\rangle\!\rangle\rangle & \overset{9}{\rightsquigarrow} \langle [\,], 16^{<}, \langle\!\langle [\kappa_{12}], o_5\rangle\!\rangle\rangle & \rightsquigarrow \langle [\,], 16, \texttt{feedDog}\rangle
\end{array}}
$$
.

This means that $\langle\!\langle [\kappa_{10}], o_2\rangle\!\rangle \rightsquigarrow \langle [\,], 16, \texttt{feedDog}\rangle$ by Rule [$\text{VF}_{\text{Trans}}$]. Finally, we can conclude that $\langle\!\langle [\kappa_{10}], o_2\rangle\!\rangle \in pt^{dd}_{cs}([\,], 16, \texttt{feedDog})$ by Rule [PT].

In addition to 1 – 9, there are other facts generated on-demand, in an (unsuccessful) attempt to identify some other objects pointed to by `feedDog`. Table 1 gives a step-by-step trace of $pt^{dd}_{cs}([\,], 16, \texttt{feedDog}) = \diamondsuit$ when operating on Figure 10, a version of Figure 7 with a complete value-flow graph for the same program. For Table 1, we would like to highlight the following three aspects:

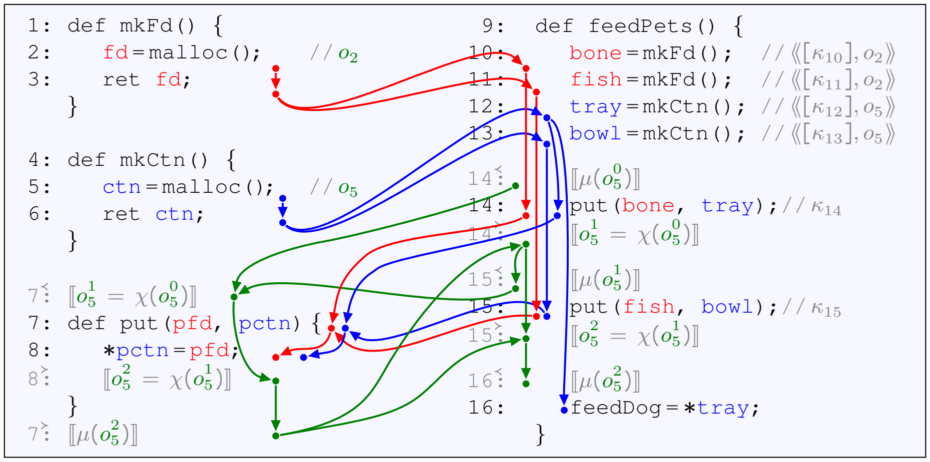



**Fig. 10.** The program given in Figure 7 decorated with all the value-flow edges.

1. **Value-Flow Transitivity.** The flow of $\langle\!\langle[\kappa_{10}], o_2\rangle\!\rangle$ into $\langle[\,], 16, \texttt{feedDog}\rangle$, i.e., $\langle\!\langle[\kappa_{10}], o_2\rangle\!\rangle \rightsquigarrow \langle[\,], 16, \texttt{feedDog}\rangle$, discussed in Example 3, is obtained by Steps #11 – #13 – #32 – #34 – #36 – #51 – #53 – #55 – #57 – #59 – #61 – #63.
2. **Generating Demand Points-to Queries.** In addition to $pt_{cs}^{dd}([\,], 16, \texttt{feedDog}) = \diamondsuit$, the other demand queries $\diamondsuit$ are issued in by firing ① Rule [$\text{DD}_{\text{Back}}$] (e.g., Steps #4, #6 and #8) to start a new backward traversal, and ② Rules [$\text{DD}_{\text{Load}}$] and [$\text{DD}_{\text{Store}}$] (e.g., Steps #2 and #19) at a load or store statement to resolve a dereferenced pointer.
3. **Context-sensitivity.** Starting with $pt_{cs}^{dd}([\,], 16, \texttt{feedDog}) = \diamondsuit$, i.e., $\diamondsuit \rightsquigarrow \langle[\,], 16, \texttt{feedDog}\rangle$ at Step #1, we obtain $\langle\!\langle[\kappa_{12}], o_5\rangle\!\rangle \rightsquigarrow \langle[\,], 16, \texttt{tray}\rangle$ in Steps #2 – #10. There are two call sites, $\kappa_{14}$ and $\kappa_{15}$, for `put()`. Once we know what `tray` points to, we can enter `put()` backwards from its exit at line $7^{\succ}$ in two ways, depending on whether it is called from $\kappa_{15}$ or $\kappa_{14}$.

By performing Steps #11 – #18 (with the assumption that `put()` is called from $\kappa_{15}$), we reach line 8, where we issue a demand query at Step #19, $\diamondsuit \rightsquigarrow \langle[\kappa_{15}], 8, \texttt{pctn}\rangle$, but only to find that $\langle\!\langle[\kappa_{13}], o_5\rangle\!\rangle \rightsquigarrow \langle[\kappa_{15}], 8, \texttt{pctn}\rangle$, i.e., $\langle\!\langle[\kappa_{12}], o_5\rangle\!\rangle \not\rightsquigarrow \langle[\kappa_{15}], 8, \texttt{pctn}\rangle$ at the end of Steps #19 – #31.

Alternatively, after having performed Steps #32 – #37 (with the assumption that `put()` is called from $\kappa_{14}$), we reach line 8 again, where we issue another query at Step #38, $\diamondsuit \rightsquigarrow \langle[\kappa_{14}], 8, \texttt{pctn}\rangle$. This time, however, we obtain $\langle\!\langle[\kappa_{12}], o_5\rangle\!\rangle \rightsquigarrow \langle[\kappa_{14}], 8, \texttt{pctn}\rangle$, i.e., at the end of Steps #38 – #50. By completing Steps #51 – #64, as already demonstrated in Example 3, we obtain $\langle\!\langle[\kappa_{10}], o_2\rangle\!\rangle \in pt_{cs}^{dd}([\,], 16, \texttt{feedDog})$.

### 4.3 Synergizing Aliasing and Control-Flow Reachability

Given a pair of deallocation $\psi(l_\psi, p)$ and dereference $\omega(l_\omega, q)$, we proceed to prove absence of $\langle\langle\!\langle c_o, o\rangle\!\rangle, \psi(c_\psi, l_\psi, p), \omega(c_\omega, l_\omega, q)\rangle$ on all the control-flow paths $\rho$ across

| Step # | $\rightsquigarrow$ | Rule | Step # | $\rightsquigarrow$ | Rule |
|---|---|---|---|---|---|
| 1 | $\diamond \rightsquigarrow \langle [\,], 16, \texttt{feedDog} \rangle$ | [QRY] | 33 | $\diamond \rightsquigarrow \langle [\,], 14^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{DD_{Back}}$] |
| 2 | $\diamond \rightsquigarrow \langle [\,], 16, \texttt{tray} \rangle$ | [$\mathrm{DD_{Load}}$] | 34 | $\langle [\kappa_{14}], 7^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle \rightsquigarrow \langle [\,], 14^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{VF^{A}_{Ret}}$] |
| 3 | $\langle [\,], 12, \texttt{tray} \rangle \rightsquigarrow \langle [\,], 16, \texttt{tray} \rangle$ | [$\mathrm{VF^{T}}$] | 35 | $\diamond \rightsquigarrow \langle [\kappa_{14}], 7^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{DD_{Back}}$] |
| 4 | $\diamond \rightsquigarrow \langle [\,], 12, \texttt{tray} \rangle$ | [$\mathrm{DD_{Back}}$] | 36 | $\langle [\kappa_{14}], 8^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle \rightsquigarrow \langle [\kappa_{14}], 7^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{VF^{A}}$] |
| 5 | $\langle [\kappa_{12}], 6, \texttt{ctn} \rangle \rightsquigarrow \langle [\,], 12, \texttt{tray} \rangle$ | [$\mathrm{VF^{T}_{Ret}}$] | 37 | $\diamond \rightsquigarrow \langle [\kappa_{14}], 8^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{DD_{Back}}$] |
| 6 | $\diamond \rightsquigarrow \langle [\kappa_{12}], 6, \texttt{ctn} \rangle$ | [$\mathrm{DD_{Back}}$] | 38 | $\diamond \rightsquigarrow \langle [\kappa_{14}], 8, \texttt{pctn} \rangle$ | [$\mathrm{DD_{Store}}$] |
| 7 | $\langle [\kappa_{12}], 5, \texttt{ctn} \rangle \rightsquigarrow \langle [\kappa_{12}], 6, \texttt{ctn} \rangle$ | [$\mathrm{VF^{T}}$] | 39 | $\langle [\kappa_{14}], 7, \texttt{pctn} \rangle \rightsquigarrow \langle [\kappa_{14}], 8, \texttt{pctn} \rangle$ | [$\mathrm{VF^{T}}$] |
| 8 | $\diamond \rightsquigarrow \langle [\kappa_{12}], 5, \texttt{ctn} \rangle$ | [$\mathrm{DD_{Back}}$] | 40 | $\diamond \rightsquigarrow \langle [\kappa_{14}], 7, \texttt{pctn} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 9 | $\langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rightsquigarrow \langle [\kappa_{12}], 5, \texttt{ctn} \rangle$ | [$\mathrm{VF_{Alloc}}$] | 41 | $\langle [\,], 14, \texttt{tray} \rangle \rightsquigarrow \langle [\kappa_{14}], 7, \texttt{pctn} \rangle$ | [$\mathrm{VF^{T}_{Call}}$] |
| 10 | $\langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rightsquigarrow \langle [\,], 16, \texttt{tray} \rangle$ | [$\mathrm{VF_{Trans}}$] | 42 | $\diamond \rightsquigarrow \langle [\,], 14, \texttt{tray} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 11 | $\langle [\,], 16^{<}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle \rightsquigarrow \langle [\,], 16, \texttt{feedDog} \rangle$ | [$\mathrm{VF_{Load}}$] | 43 | $\langle [\,], 12, \texttt{tray} \rangle \rightsquigarrow \langle [\,], 14, \texttt{tray} \rangle$ | [$\mathrm{VF^{T}}$] |
| 12 | $\diamond \rightsquigarrow \langle [\,], 16^{<}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{DD_{Back}}$] | 44 | $\diamond \rightsquigarrow \langle [\,], 12, \texttt{tray} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 13 | $\langle [\,], 15^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle \rightsquigarrow \langle [\,], 16^{<}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{VF^{A}}$] | 45 | $\langle [\kappa_{12}], 6, \texttt{ctn} \rangle \rightsquigarrow \langle [\,], 12, \texttt{tray} \rangle$ | [$\mathrm{VF^{T}_{Ret}}$] |
| 14 | $\diamond \rightsquigarrow \langle [\,], 15^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{DD_{Back}}$] | 46 | $\diamond \rightsquigarrow \langle [\kappa_{12}], 6, \texttt{ctn} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 15 | $\langle [\kappa_{15}], 7^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle \rightsquigarrow \langle [\,], 15^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{VF^{A}_{Ret}}$] | 47 | $\langle [\kappa_{12}], 5, \texttt{ctn} \rangle \rightsquigarrow \langle [\kappa_{12}], 6, \texttt{ctn} \rangle$ | [$\mathrm{VF^{T}}$] |
| 16 | $\diamond \rightsquigarrow \langle [\kappa_{15}], 7^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{DD_{Back}}$] | 48 | $\diamond \rightsquigarrow \langle [\kappa_{12}], 5, \texttt{ctn} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 17 | $\langle [\kappa_{15}], 8^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle \rightsquigarrow \langle [\kappa_{15}], 7^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{VF^{A}}$] | 49 | $\langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rightsquigarrow \langle [\kappa_{12}], 5, \texttt{ctn} \rangle$ | [$\mathrm{VF_{Alloc}}$] |
| 18 | $\diamond \rightsquigarrow \langle [\kappa_{15}], 8^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{DD_{Back}}$] | 50 | $\langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rightsquigarrow \langle [\kappa_{14}], 8, \texttt{pctn} \rangle$ | [$\mathrm{VF_{Trans}}$] |
| 19 | $\diamond \rightsquigarrow \langle [\kappa_{15}], 8, \texttt{pctn} \rangle$ | [$\mathrm{DD_{Store}}$] | 51 | $\langle [\kappa_{14}], 8, \texttt{pfd} \rangle \rightsquigarrow \langle [\kappa_{14}], 8^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{VF_{Store}}$] |
| 20 | $\langle [\kappa_{15}], 7, \texttt{pctn} \rangle \rightsquigarrow \langle [\kappa_{15}], 8, \texttt{pctn} \rangle$ | [$\mathrm{VF^{T}}$] | 52 | $\diamond \rightsquigarrow \langle [\kappa_{14}], 8, \texttt{pfd} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 21 | $\diamond \rightsquigarrow \langle [\kappa_{15}], 7, \texttt{pctn} \rangle$ | [$\mathrm{DD_{Back}}$] | 53 | $\langle [\kappa_{14}], 7, \texttt{pfd} \rangle \rightsquigarrow \langle [\kappa_{14}], 8, \texttt{pfd} \rangle$ | [$\mathrm{VF^{T}}$] |
| 22 | $\langle [\,], 15, \texttt{bowl} \rangle \rightsquigarrow \langle [\kappa_{15}], 7, \texttt{pctn} \rangle$ | [$\mathrm{VF^{T}_{Call}}$] | 54 | $\diamond \rightsquigarrow \langle [\kappa_{14}], 7, \texttt{pfd} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 23 | $\diamond \rightsquigarrow \langle [\,], 15, \texttt{bowl} \rangle$ | [$\mathrm{DD_{Back}}$] | 55 | $\langle [\,], 14, \texttt{bone} \rangle \rightsquigarrow \langle [\kappa_{14}], 7, \texttt{pfd} \rangle$ | [$\mathrm{VF^{T}_{Call}}$] |
| 24 | $\langle [\,], 13, \texttt{bowl} \rangle \rightsquigarrow \langle [\,], 15, \texttt{bowl} \rangle$ | [$\mathrm{VF^{T}}$] | 56 | $\diamond \rightsquigarrow \langle [\,], 14, \texttt{bone} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 25 | $\diamond \rightsquigarrow \langle [\,], 13, \texttt{bowl} \rangle$ | [$\mathrm{DD_{Back}}$] | 57 | $\langle [\,], 10, \texttt{bone} \rangle \rightsquigarrow \langle [\,], 14, \texttt{bone} \rangle$ | [$\mathrm{VF^{T}}$] |
| 26 | $\langle [\kappa_{13}], 6, \texttt{ctn} \rangle \rightsquigarrow \langle [\,], 13, \texttt{bowl} \rangle$ | [$\mathrm{VF^{T}_{Ret}}$] | 58 | $\diamond \rightsquigarrow \langle [\,], 10, \texttt{bone} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 27 | $\diamond \rightsquigarrow \langle [\kappa_{13}], 6, \texttt{ctn} \rangle$ | [$\mathrm{DD_{Back}}$] | 59 | $\langle [\kappa_{10}], 3, \texttt{fd} \rangle \rightsquigarrow \langle [\,], 10, \texttt{bone} \rangle$ | [$\mathrm{VF^{T}_{Ret}}$] |
| 28 | $\langle [\kappa_{13}], 5, \texttt{ctn} \rangle \rightsquigarrow \langle [\kappa_{13}], 6, \texttt{ctn} \rangle$ | [$\mathrm{VF^{T}}$] | 60 | $\diamond \rightsquigarrow \langle [\kappa_{10}], 3, \texttt{fd} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 29 | $\diamond \rightsquigarrow \langle [\kappa_{13}], 5, \texttt{ctn} \rangle$ | [$\mathrm{DD_{Back}}$] | 61 | $\langle [\kappa_{10}], 2, \texttt{fd} \rangle \rightsquigarrow \langle [\kappa_{10}], 3, \texttt{fd} \rangle$ | [$\mathrm{VF^{T}}$] |
| 30 | $\langle\!\langle [\kappa_{13}], o_5 \rangle\!\rangle \rightsquigarrow \langle [\kappa_{13}], 5, \texttt{ctn} \rangle$ | [$\mathrm{VF_{Alloc}}$] | 62 | $\diamond \rightsquigarrow \langle [\kappa_{10}], 2, \texttt{fd} \rangle$ | [$\mathrm{DD_{Back}}$] |
| 31 | $\langle\!\langle [\kappa_{13}], o_5 \rangle\!\rangle \rightsquigarrow \langle [\kappa_{15}], 8, \texttt{pctn} \rangle$ | [$\mathrm{VF_{Trans}}$] | 63 | $\langle\!\langle [\kappa_{10}], o_2 \rangle\!\rangle \rightsquigarrow \langle [\kappa_{10}], 2, \texttt{fd} \rangle$ | [$\mathrm{VF_{Alloc}}$] |
| 32 | $\langle [\,], 14^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle \rightsquigarrow \langle [\,], 15^{>}, \langle\!\langle [\kappa_{12}], o_5 \rangle\!\rangle \rangle$ | [$\mathrm{VF^{A}}$] | 64 | $\langle\!\langle [\kappa_{10}], o_2 \rangle\!\rangle \rightsquigarrow \langle [\,], 16, \texttt{feedDog} \rangle$ | [$\mathrm{VF_{Trans}}$] |

**Table 1.** A step-by-step trace of $pt^{dd}_{cs}([\,], 16, \texttt{feedDog}) = \diamond$, for computing $\langle\!\langle [\kappa_{10}], o_2 \rangle\!\rangle \in pt^{dd}_{cs}([\,], 16, \texttt{feedDog})$, with $pt^{dd}_{cs}$ operating on the value-flow graph of the program in Figure 10 by applying the rules given in Figure 9.

the ICFG of the program, where $c_\psi \in C_\psi$ and $c_\omega \in C_\omega$ are calling contexts for $l_\psi$ and $l_\omega$, respectively. We abstract $\rho$ with a context tuple $(c_o, c_\psi, c_\omega)$, which is shortened to $(c_\psi, c_\omega)$, since $c_o$ can be automatically inferred by $pt^{dd}_{cs}$ from $c_\psi$ and $c_\omega$.

The following two properties are checked context-sensitively:

- **Aliasing**, $\mathcal{A}^{\psi}_{\omega} : C_\psi \times C_\omega \to \{\mathit{true}, \mathit{false}\}$, indicating if $(c_\psi, p)$ aliases $(c_\omega, q)$, and

$$[\text{AliasingAndReaching}]\frac{\mathcal{R}^{\psi}_{\omega}(c_\psi, c_\omega) \quad \mathcal{A}^{\psi}_{\omega}(c_\psi, c_\omega)}{\mathcal{S}^{\psi}_{\omega}(c_\psi, c_\omega)}$$

$$[\text{Aliasing}]\frac{\begin{array}{c} pt^{dd}_{cs}(c_\psi, l_\psi, p) = \diamondsuit \;\vdash\; \langle\!\langle hc_\psi, o\rangle\!\rangle \in pt(c_\psi, l_\psi, p) \\ pt^{dd}_{cs}(c_\omega, l_\omega, q) = \diamondsuit \;\vdash\; \langle\!\langle hc_\omega, o\rangle\!\rangle \in pt(c_\omega, l_\omega, q) \\ cons(\_, hc_\psi) = hc_\omega \;\vee\; cons(\_, hc_\omega) = hc_\psi \end{array}}{\mathcal{A}^{\psi}_{\omega}(c_\psi, c_\omega)}$$

$$[\text{Reaching}]\frac{\overline{l_\psi} = car(cons(c_\psi, l_\psi)) \quad \overline{l_\omega} = car(cons(c_\omega, l_\omega)) \quad \mathcal{R}_{Intra}(\overline{l_\psi}, \overline{l_\omega})}{\mathcal{R}^{\psi}_{\omega}(c_\psi, c_\omega)}$$

**Fig. 11.** Rules for synergizing aliasing and control-flow reachability.

- **Reachability**, $\mathcal{R}^{\psi}_{\omega} : C_\psi \times C_\omega \rightarrow \{true, false\}$, indicating if $l_\psi$ reaches $l_\omega$ on the ICFG by going through first the return edges specified by $c_\psi$ and then the call edges specified by $c_\omega$.

We consider aliasing and reachability together, $\mathcal{S}^{\psi}_{\omega} : C_\psi \times C_\omega \rightarrow \{true, false\}$, by requiring $\mathcal{A}^{\psi}_{\omega}$ and $\mathcal{R}^{\psi}_{\omega}$ to be satisfied for the same context pair $(c_\psi, c_\omega)$. We report a TH-safety violation at the dereference iff $\mathcal{S}^{\psi}_{\omega}$ is satisfied, thereby avoiding false-positives that satisfy both constraints on two different paths only.

Figure 11 gives our rules. Rule [Aliasing] computes an abstract path, $(c_\psi, c_\omega)$, on which $p$ aliases $q$. Note that $\langle\!\langle hc_\psi, o\rangle\!\rangle$ and $\langle\!\langle hc_\omega, o\rangle\!\rangle$ may represent the same (concrete) object if one of these two contexts is a suffix of (i.e., coarser than) the other. Rule [Reaching] computes an abstract path, $(c_\psi, c_\omega)$, on which $l_\psi$ reaches $l_\omega$, which happens if $l_\psi$ first reaches $\overline{l_\psi}$ inter-procedurally via the return edges specified by $c_\psi$, then $\overline{l_\psi}$ reaches $\overline{l_\omega}$ intra-procedurally in the same function (denoted $\mathcal{R}_{Intra}(\overline{l_\psi}, \overline{l_\omega})$), and finally, $\overline{l_\omega}$ reaches $l_\omega$ inter-procedurally via the call edges specified by $c_\omega$.

### 4.4 Adaptive Context-Sensitivity

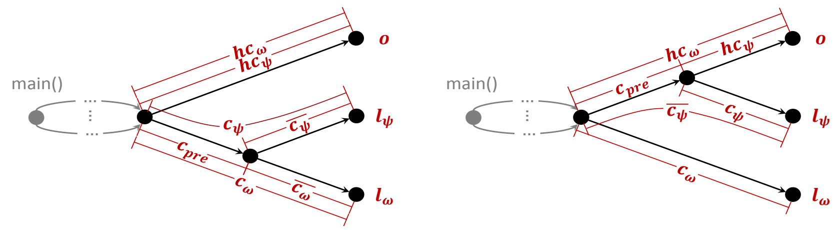
To guarantee soundness, all context pairs $(c_\psi, c_\omega) \in C_\psi \times C_\omega$ in the program must be considered, making [Aliasing] in Figure 11 prohibitively costly to verify. To tame path explosion, we use the two rules in Figure 12 instead with adaptive context-sensitivity, thereby reducing significantly the number of context pairs considered without losing soundness or precision. We explain these two rules, illustrated in Figure 13, below.

The key insight behind is that $pt^{dd}_{cs}([\,], l, v)$, when asked to compute the points-to set of $(l, v)$ with an empty context $[\,]$, which represents an abstraction of all possible contexts (from `main()`), will return $\langle\!\langle hc, o\rangle\!\rangle \in pt^{dd}_{cs}(c, l, v)$, where the contexts $c$ and $hc$ are automatically inferred. In particular, $c$ and $hc$ are appropriately $k$-limited (with

$$[\text{Aliasing-EqHeapCtx}]\ \frac{\begin{array}{ll} pt_{cs}^{dd}([\,], l_\psi, p) = \diamond \ \vdash \ \langle\!\langle hc, o\rangle\!\rangle \in pt_{cs}^{dd}(c_\psi, l_\psi, p) & c_\psi = cons(c_{pre}, \overline{c_\psi}) \\ pt_{cs}^{dd}([\,], l_\omega, q) = \diamond \ \vdash \ \langle\!\langle hc, o\rangle\!\rangle \in pt_{cs}^{dd}(c_\omega, l_\omega, q) & c_\omega = cons(c_{pre}, \overline{c_\omega}) \end{array}}{\mathcal{A}_\omega^\psi(\overline{c_\psi}, \overline{c_\omega})}$$

$$[\text{Aliasing-NeqHeapCtx}]\ \frac{\begin{array}{ll} pt_{cs}^{dd}([\,], l_\psi, p) = \diamond \ \vdash \ \langle\!\langle hc_\psi, o\rangle\!\rangle \in pt_{cs}^{dd}(c_\psi, l_\psi, p) & \overline{c_\psi} = cons(c_{pre}, c_\psi) \\ pt_{cs}^{dd}([\,], l_\omega, q) = \diamond \ \vdash \ \langle\!\langle hc_\omega, o\rangle\!\rangle \in pt_{cs}^{dd}(c_\omega, l_\omega, q) & hc_\omega = cons(c_{pre}, hc_\psi) \end{array}}{\mathcal{A}_\omega^\psi(\overline{c_\psi}, c_\omega)}$$

**Fig. 12.** Two rules for replacing [Aliasing] in Figure 11 with adaptive context-sensitivity.




(a) [Aliasing-EqHeapCtx] ($hc_\psi = hc_\omega$) (b) [Aliasing-NeqHeapCtx] ($hc_\psi \neq hc_\omega$)

**Fig. 13.** An illustration of the two rules in Figure 12, where a fat dot represents a function and an arrow represents a sequence of (transitive) function calls across the functions in the program.

any unnecessary context prefix $c_{pre}$ from `main()` truncated), since we have:

$$pt_{cs}^{dd}([\,], l, v) = \diamond \ \vdash \ \langle\!\langle hc, o\rangle\!\rangle \in pt_{cs}^{dd}(c, l, v) \iff \langle\!\langle cons(c_{pre}, hc), o\rangle\!\rangle \in pt_{cs}^{dd}(cons(c_{pre}, c), l, v)$$

In [Aliasing], there are three possibilities for $\langle\!\langle hc_\psi, o\rangle\!\rangle$ and $\langle\!\langle hc_\omega, o\rangle\!\rangle$ to be aliases:

1. $hc = hc_\psi = hc_\omega$**.** This case, illustrated in Figure 13(a), is handled by [Aliasing-EqHeapCtx], which says that it suffices to consider only $(\overline{c_\psi}, \overline{c_\omega})$ by removing any common prefix $c_{pre}$ from $c_\psi$ and $c_\omega$, since $(\overline{c_\psi}, \overline{c_\omega})$ is coarser than $(c_\psi, c_\omega)$. In addition, all context pairs $(cons(c_{pre}^1, c_\psi), cons(c_{pre}^2, c_\omega))$, where $c_{pre}^1 \neq c_{pre}^2$, can also be soundly removed, since $\langle\!\langle cons(c_{pre}^1, hc), o\rangle\!\rangle$ cannot be aliased with $\langle\!\langle cons(c_{pre}^2, hc), o\rangle\!\rangle$. By construction, $car(cons(\overline{c_\psi}, l_\psi))$ and $car(cons(\overline{c_\omega}, l_\omega))$ are guaranteed to be in the same function, allowing $\mathcal{R}_\omega^\psi$ in [Reaching] to be checked trivially.
2. $hc_\omega = cons(c, hc_\psi)$**.** To check $\mathcal{R}_\omega^\psi$ in [Reaching] efficiently, [Aliasing-NeqHeapCtx], as shown in Figure 13(b), constructs $\overline{c_\psi}$ by extending $c_\psi$ such that $car(cons(\overline{c_\psi}, l_\psi))$ and $car(cons(c_\omega, l_\omega))$ reside in the same function. As in [Aliasing-EqHeapCtx], all context-pairs $(cons(c_{pre}^1, \overline{c_\psi}), cons(c_{pre}^2, c_\omega))$, where $c_{pre}^1 \neq c_{pre}^2$, are ignored

soundly. In addition, $car(cons(\overline{c_\psi}, l_\psi))$ and $car(cons(c_\omega, l_\omega))$ always reside in the same function, allowing $\mathcal{R}_\omega^\psi$ in [Reaching] to be checked trivially as above.

3. $hc_\psi = cons(c, hc_\omega)$. This case, which indicates a use-before-free, is always safe.

Our approach $\mathrm{D}^3$ is adaptive since its search space exploration selects calling contexts with appropriate lengths adaptively without losing soundness or precision.

**Example 4.** Let us apply our rules to the program in Figure 4(a) to detect the TH-safety violation $\langle\langle\!\langle[\kappa_{18}, \kappa_{21}], o_2\rangle\!\rangle, \psi([\kappa_{18}, \kappa_{22}], l_5, \mathrm{y}), \omega([\kappa_{19}], l_7, \mathrm{z})\rangle$. Let us consider [Aliasing-NeqHeapCtx] first. For the two points-to queries $pt_{cs}^{dd}([\,], l_5, \mathrm{y}) = \diamondsuit$ and $pt_{cs}^{dd}([\,], l_7, \mathrm{z}) = \diamondsuit$ issued, we obtain $\langle\!\langle[\kappa_{21}], o_2\rangle\!\rangle \in pt_{cs}^{dd}([\kappa_{22}], l_5, \mathrm{y})$ and $\langle\!\langle[\kappa_{18}, \kappa_{21}], o_2\rangle\!\rangle \in pt_{cs}^{dd}([\kappa_{19}], l_7, \mathrm{z})$. As $hc_\omega = [\kappa_{18}, \kappa_{21}] = cons([\kappa_{18}], [\kappa_{21}]) = cons(c_{pre}, hc_\psi)$, we have $\overline{c_\psi} = cons(c_{pre}, c_\psi) = [\kappa_{18}, \kappa_{21}]$. By applying [Aliasing-NeqHeapCtx], $\mathcal{A}_\omega^\psi([\kappa_{18}, \kappa_{21}], [\kappa_{19}])$ holds. Let $\overline{l_\psi} = \kappa_{18}$ and $\overline{l_\omega} = \kappa_{19}$. By applying [Reaching], $\mathcal{R}_\omega^\psi([\kappa_{18}, \kappa_{21}], [\kappa_{19}])$ holds. Finally, by [AliasingAndReaching], $\mathcal{S}([\kappa_{18}, \kappa_{21}], [\kappa_{19}])$ holds, triggering this as a TH-safety violation.

### 4.5 Soundness

For a program $P$ considered in Section 2, $\mathrm{D}^3$ (Figure 5) is sound. First, $G_{\mathrm{vfg}}$ constructed for $P$, based on the rules in Figure 8, over-approximates the flow of any value in $P$ as Andersen's analysis (Figure 2) is sound. Second, $pt_{cs}^{dd}$ (Figure 9) is sound as it over-approximates the points-to information in $P$. Third, we suppress a TH-safety violation warning soundly according to [AliasingAndReaching] (Figure 11). Finally, our adaptive analysis (Figure 12) is sound as the context pairs $(c_\psi, c_\omega)$ pruned for [AliasingAndReaching] during the search space exploration are redundant (Section 4.4).

## 5 Evaluation

We show that $\mathrm{D}^3$ can accomplish our TH-safety verification task for reasonably large C programs efficiently with good precision in the context of the prior work.

### 5.1 Methodology

We have implemented $\mathrm{D}^3$ in the open-source program analysis framework, SVF [50], which is implemented in LLVM [27]. Given a program, its source files are first compiled individually into LLVM IR by the Clang compiler front-end, before linked together into a single whole-program IR file by the LLVM Gold Plugin. Our TH-safety verification task is then performed statically on the whole-program LLVM IR file.

Two sets of benchmark are used. One set consists of 138 test cases with the ground truth for use-after-free vulnerabilities (CWE-416) from the NIST Juliet Test Suite for C [1], which are all TH-safety violations extracted from real-world scenarios, with one per test case. The other set consists of ten popular open-source C programs (with 40 – 260 KLOC) given in Table 2, containing a total of 114,508 pointer dereferences.

| Program | Characteristics | | Value-Flow Graph | | $\mathrm{D}^{\mathrm{SEP}}$ | | | $\mathrm{D}^3$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | KLOC | #Derefs | #Nodes | #Edges | Time (s) | #Safe | %Safe | Time (s) | #Safe | %Safe | %Impr |
| a2ps-4.14 | 65 | 12,601 | 35,201 | 58,255 | 428 | 7,000 | 55.6% | 5,653 | 9,944 | 78.9% | 52.6% |
| cpio-2.12 | 94 | 5,211 | 13,486 | 23,379 | 10 | 3,805 | 73.0% | 180 | 4,964 | 95.3% | 82.4% |
| ctags-5.8 | 42 | 14,628 | 56,320 | 152,846 | 54 | 10,538 | 72.0% | 520 | 14,014 | 95.8% | 85.0% |
| MCSim-6.0.1 | 60 | 8,718 | 17,914 | 28,365 | 64 | 5,233 | 60.0% | 1,010 | 8,105 | 93.0% | 82.4% |
| parted-3.2 | 138 | 1,493 | 7,703 | 16,415 | 9 | 1,133 | 75.9% | 14 | 1,371 | 91.8% | 66.1% |
| patch-2.7.6 | 88 | 5,334 | 16,926 | 35,269 | 50 | 4,065 | 76.2% | 480 | 4,961 | 93.0% | 70.6% |
| sendmail-8.15 | 260 | 21,536 | 128,312 | 328,892 | 1,332 | 12,368 | 57.4% | 3,277 | 15,570 | 72.3% | 34.9% |
| tar-1.31 | 191 | 11,671 | 54,594 | 109,269 | 225 | 7,741 | 66.3% | 7,672 | 9,200 | 78.8% | 37.1% |
| tmux-2.8 | 54 | 24,877 | 91,373 | 185,594 | 166 | 12,366 | 49.7% | 12,295 | 18,266 | 73.4% | 47.2% |
| wget-1.20 | 174 | 8,439 | 31,460 | 63,738 | 100 | 5,957 | 70.6% | 1,920 | 6,746 | 79.9% | 31.8% |
| Avg. | 117 | 11,451 | 45,329 | 100,202 | 244 | 7,021 | 65.7% | 3,302 | 9,314 | 85.2% | 59.0% |
| Total | 1,166 | 114,508 | 453,289 | 1,002,022 | 2,438 | 70,206 | 61.3% | 33,022 | 93,141 | 81.3% | 51.8% |

**Table 2.** Results for verifying 10 open-source C programs. $\mathrm{D}^{\mathrm{SEP}}$ is a version of $\mathrm{D}^3$ with aliasing $\mathcal{A}_\omega^\psi$ and reachability $\mathcal{R}_\omega^\psi$ checked separately. %Impr is computed as $\frac{\mathrm{D}^3.\#\mathrm{Safe}-\mathrm{D}^{\mathrm{SEP}}.\#\mathrm{Safe}}{\#\mathrm{Deref}-\mathrm{D}^{\mathrm{SEP}}.\#\mathrm{Safe}} \times 100\%$.

We compare $\mathrm{D}^3$ with a C bounded model checker, CBMC (version 5.11) [26]. CBMC, as confirmed by the authors, does not provide an option to verify TH-safety only by disabling other types of memory errors. Thus, we have configured it with the "pointercheck" option to detect all pointer-related errors and then manually extracted all the TH-safety violations reported. For the small test cases in the NIST Juliet Test Suite, loops are not bounded. For the ten real-world programs, loops are unwound by using "unwind 2" to accelerate termination (at the expense of losing soundness).

Infer [7] (i.e., Abductor earlier [8]) has evolved into a bug detector by sacrificing soundness, with its older verification-oriented versions no longer available (as confirmed by its authors), The latest version of SLAyer [6] does not compile (as also confirmed by its authors) since it relies on a specific yet unknown old subversion of the Z3 SMT-solver. So we will not compare with such separation-logic-based verifiers, as Infer, for example, is now designed to lower its false positive rate by tolerating for false negatives.

In addition, we also evaluate $\mathrm{D}^3$ against a version of $\mathrm{D}^3$, denoted $\mathrm{D}^{\mathrm{SEP}}$, for which aliasing and control-flow reachability are considered separately.

As $pt_{cs}^{dd}$ is demand-driven, the time budget for a points-to query issued from [Aliasing] (Figure 12) is set to be a maximum of 10,000 value-flow edges traversed. On time out, $pt_{cs}^{dd}$ will fall back to the result computed by Andersen's pointer analysis, $pt$, soundly (Figure 2). We have done our experiments on a machine with a 3.5 GHz Intel Xeon 16-core CPU and 256 GB memory, running Ubuntu OS (version 16.04 LTS). The analysis time of a program is the average of five runs. For $\mathrm{D}^3/\mathrm{D}^{\mathrm{SEP}}$, the analysis times from all its stages (Figure 5) are included, except the pre-analysis, since Andersen's analysis is expected to be reused by many other static analyses for the program.

## 5.2 Results and Analysis

**5.2.1 Juliet Test Suite: Soundness** Both CBMC and $D^3$ report soundly all the 138 use-after-free bugs without any false positives. Each test case is small, with a few hundreds of LOC, costing less than one second to verify by either tool.

**5.2.2 The Ten Open-Source Programs: Precision and Scalability** For any of these programs, CBMC, which is bounded by even "unwind 2", cannot terminate within a 1-day time budget. We have decided to evaluate $D^3$ against a version, $D^{SEP}$, in which both aliasing and control-flow reachability are considered separately, as shown in Table 2.

- **Precision.** For a total of 114,508 dereferences in the ten programs, $D^3$ proves successfully 81.3% (or $\frac{93{,}141}{114{,}508}$) to be safe. This translates into an average of 85.2% per program, ranging from 72.3% in `sendmail` to 95.8% for `ctags`. For the remaining 14.8%, anout an average of 33% fail due to the out-of-budget problem. In contrast, $D^{SEP}$ finds only 61.3% of all the dereferences to be safe, with an average of 65.7% per program, ranging from 49.7% for `tmux` to 76.2% for `patch`.

  $D^3$ is significantly more precise than $D^{SEP}$ (as measured by %Impr). For a total of 44,302 dereferences that cannot be verified to be safe by $D^{SEP}$, $D^3$ recognizes 51.8% of these (i.e., $\frac{22{,}935}{44{,}302}$) as being safe. The largest improvements are observed for `ctags` (85.0%), `cpio` (82.4%) and `MCSim` (82.4%), which contain many cases as illustrated in Figure 3, causing $D^{SEP}$ to fail but $D^3$ to succeed, since aliasing and reachability must be considered together. On the other hand, the precision improvements for `wget` (31.8%) and `sendmail` (34.9%), where linked lists are heavily used, are the least impressive.
- **Scalability.** For a given program, the size of its value-flow graph affects the time complexity of our approach. $D^3$ scales reasonably well to these programs, spending a total of 33,022 seconds on analyzing a total of 1,166 KLOC, while $D^{SEP}$ is faster (finishing in 2,438 seconds) but less precise. For `sendmail` (the largest with 260 KLOC), $D^3$ takes 3,277 seconds to complete. For `ctags` (the smallest with 42 KLOC), $D^3$ finishes in 520 seconds. $D^3$ is the fastest for `parted`, which has the smallest value-flow graph with the smallest number of dereferences. $D^3$ is the slowest for `tmux`, which has the second largest value-flow graph with the largest number of dereferences.

# 6 Related Work

***Pointer Analysis*** Substantial progress has been made for whole-program [33,23,48] and demand-driven [20,47,51] pointer analyses, with flow-sensitivity [19,31], call-site-sensitivity [40,61], object-sensitivity [37,55] and type-sensitivity [45,25]). These recent advances in both precision and scalability have resulted in their widespread adoption in detecting memory bugs [2,17], such as memory leaks [9,52], null dereferences [36,34], uninitialized variables [60,35], buffer overflows [30,10], and typestate verification [16,12].

Pointer-analysis-based tools [44,57] can detect TH-safety violations with low false-positive rates, but at the expense of missing true bugs. Some recent advances on pointer analysis for object-oriented languages [32,46] improve the efficiency of the traditional $k$-object-sensitivity by analyzing some methods context-insensitively, but due to the lack of flow-insensitivity, such techniques are unsuitable for analyzing TH-safety. In contrast, $D^3$ is designed to be a verifier for finding TH-safety violations with good precision soundly by considering aliasing and control-flow reachability synergistically.

***Separation Logic*** As an extension of Hoare logic for heap-manipulating programs, separation logic [42] provides the basis for a long line of research on memory safety verification. At its core is the separating conjunction $*$ that splits the heap into disjoint heaplets, allowing program reasoning to be confined in heaplets [59,15]. For separation-logic-based verification, scalability has considerably improved with techniques like bi-abduction at the expense of sacrificing some precision [8,58], leading to industrial-strength tools such as Microsoft's SLAyer [6] and Facebook's Infer [7]. By giving up also some soundness, many industrial-strength static analyzers, such as Clang Static Analyzer [3,43] and Infer (the current release 0.15.0) are bug detectors, which reduce false positives at the expense of exhibiting false negatives as well. Unlike separation-logic-based approaches that support compositional and modular reasoning, $D^3$ takes a pointer-analysis-based approach by analyzing also only the relevant code on-demand.

***Model Checking*** Model checking represents a powerful framework for reasoning about a wide range of properties [24]. To analyze pointer-intensive C programs, model checkers such as SLAM [5] and BLAST [21] rely on pre-computed points-to information. Goal-driven techniques like SMACK+Corral [18] aim at improving scalability by simplifying verification conditions. However, as pointed out in [26], model checking still suffers from limitations in fully automated TH-safety verification for large-sized programs, partly due to complex pointer aliasing. Model checkers with symbolic execution (e.g., Symbiotic [53]) can find bugs precisely but with limited scalability for large-sized programs due to path explosion.

## 7 Conclusion

This paper presents $D^3$, a novel approach for addressing the TH-safety verification problem based on a demand-driven context-sensitive pointer analysis. $D^3$ achieves its precision (by considering both aliasing and control-flow reachability simultaneously) and scalability (with adaptive context-sensitivity). In future work, we plan to empower $D^3$ by also considering (partial) path-sensitivity and shape analysis.

## 8 Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments. This research is supported by an Australian Research Grant DP180104169.

## References


1. Juliet Test Suite 1.2. https://samate.nist.gov/srd/testsuite.php.
2. Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *PASTE '07*, pages 43–48, 2007.
3. Clang Static Analyzer. http://clang-analyzer.llvm.org/.
4. Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, 1994.
5. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K Rajamani. Automatic predicate abstraction of c programs. In *PLDI '01*, pages 203–213, 2001.
6. Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV '11*, pages 178–183, 2011.
7. Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA FM '11*, pages 459–465, 2011.
8. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL '09*, pages 289–300, 2009.
9. Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07*, pages 480–491, 2007.
10. Cristina Cifuentes, Nathan Keynes, Lian Li, Nathan Hawes, Manuel Valdiviezo, Andrew Browne, Jacob Zimmermann, Andrew Craik, Douglas Teoh, and Christian Hoermann. Static deep error checking in large system applications using parfait. In *ESEC/FSE '11*, pages 432–435, 2011.
11. Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.
12. Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI '02*, pages 57–68, 2002.
13. Isil Dillig and Thomas Dillig. Explain: A tool for performing abductive inference. In *CAV '13*, pages 684–689, 2013.
14. Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI '08*, pages 270–280, 2008.
15. Dino Distefano, Peter W Ohearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS '06*, pages 287–302, 2006.
16. Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2008.
17. Brian Hackett and Alex Aiken. How is aliasing used in systems software? In *FSE '06*, pages 69–80, 2006.
18. Arvind Haran, Montgomery Carter, Michael Emmi, Akash Lal, Shaz Qadeer, and Zvonimir Rakamari. Smack+corral: A modular verifier. In *TACAS '15*, pages 451–454, 2015.
19. Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09*, pages 226–238, 2009.
20. Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *PLDI '01*, pages 24–34, 2001.
21. Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L McMillan. Abstractions from proofs. In *POPL '04*, pages 232–244, 2004.
22. Thomas A Henzinger, George C Necula, Ranjit Jhala, Gregoire Sutre, Rupak Majumdar, and Westley Weimer. Temporal-safety proofs for systems code. In *CAV '02*, pages 526–538, 2002.

23. Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. In *OOPSLA '14*, pages 100:1–100:28, 2017.
24. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21, 2009.
25. George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI '13*, pages 423–434, 2013.
26. Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *TACAS '14*, pages 389–391, 2014.
27. Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–86, 2004.
28. Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS '15*, 2015.
29. Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL '11*, pages 3–16, 2011.
30. Lian Li, Cristina Cifuentes, and Nathan Keynes. Practical and effective symbolic analysis for buffer overflow detection. In *FSE '10*, pages 317–326, 2010.
31. Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *ESEC/FSE '11*, pages 343–353, 2011.
32. Yue Li, Tian Tan, Anders Mller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. In *OOPSLA '18*, page 141, 2018.
33. Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *POPL '11*, pages 31–42, 2011.
34. Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA '08*, pages 213–224, 2008.
35. Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *CCS '16*, pages 920–932, 2016.
36. Ravichandhran Madhavan and Raghavan Komondoor. Null dereference verification via over-approximated weakest pre-conditions analysis. In *OOSPLA '11*, pages 1033–1052, 2011.
37. Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.
38. Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. Cmc: A pragmatic approach to model checking real code. In *OSDI '02*, pages 75–88, 2002.
39. Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. In *ISMM'10*, pages 31–40, 2010.
40. Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *PLDI '14*, pages 475–484, 2014.
41. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*, pages 49–61, 1995.
42. John C Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02*, pages 55–74, 2002.
43. Coverity Scan. https://scan.coverity.com/.
44. Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *PLDI '18*, pages 693–706, 2018.
45. Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL '11*, pages 17–30, 2011.
46. Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *PLDI '14*, pages 485–495, 2014.

47. Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *ECOOP '16*, pages 22:1–22:26, 2016.
48. Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI '16*, pages 387–400, 2006.
49. Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *FSE '16*, pages 460–473, 2016.
50. Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *CC '16*, pages 265–266, 2016.
51. Yulei Sui and Jingling Xue. Value-flow-based demand-driven pointer analysis for c and c++. *IEEE Transactions on Software Engineering (TSE)*, 2018.
52. Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264, 2012.
53. Symbiotic. https://github.com/staticafi/symbiotic.
54. Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *SP '13*, pages 48–62, 2013.
55. Tian Tan, Yue Li, and Jingling Xue. Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata. In *PLDI '17*, pages 278–291, 2017.
56. Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *ACSAC '17*, pages 42–54, 2017.
57. Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *ICSE '18*, pages 327–337, 2018.
58. Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter OHearn. Scalable shape analysis for systems code. In *CAV '08*, pages 385–398, 2008.
59. Hongseok Yang and Peter OHearn. A semantic basis for local reasoning. In *FoSSaCS '02*, pages 402–416, 2002.
60. Ding Ye, Yulei Sui, and Jingling Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *CGO '14*, pages 154–164, 2014.
61. Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by level: Making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*, pages 218–229, 2010.
62. Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *PLDI '14*, pages 239–248, 2014.