Spatio-Temporal Context Reduction: A Pointer-Analysis-based Static Approach for Detecting Use-After-Free Vulnerabilities

Anonymous Author(s)

ABSTRACT

Zero-day Use-After-Free (UAF) vulnerabilities are increasingly popular and highly dangerous, but few mitigations exist. We introduce a new pointer-analysis-based static analysis, Stc, for finding UAF bugs in multi-MLOC C programs efficiently and effectively. Stc achieves this by making three advances: (i) a spatio-temporal context reduction technique for scaling down soundly and precisely the exponential number of contexts that would otherwise be considered at a pair of free and use sites, (ii) a multi-stage analysis for filtering out false alarms efficiently, and (iii) a path-sensitive demand-driven approach for finding the points-to information required.

We have implemented Stc in LLVM-3.8.0 and compared it with four different state-of-the-art static tools: CBMC (model checking), Clang (abstract interpretation), Coccinelle (pattern matching), and Supa (pointer analysis) using all the C test cases in Juliet Test Suite (JTS) and 10 open-source C applications. For the ground-truth validated with JTS, Stc detects all the 138 known UAF bugs as CBMC and Supa do while Clang and Coccinelle misses some bugs, with no false alarms from any tool. For practicality validated with the 10 applications (totaling 3+ MLOC), Stc reports 132 warnings including 85 bugs in 7.6 hours while the existing tools are either unscalable by terminating within 3 days only for one application (CBMC) or impractical by finding virtually no bugs (Clang and Coccinelle) or issuing an excessive number of false alarms (Supa).

1 INTRODUCTION

Use-After-Free (UAF) vulnerabilities, i.e., dangling pointer dereferences (referencing an object that has been freed), are increasingly being exploited, as shown in Figure 1. UAF vulnerabilities are highly dangerous, with 80.14% in the NVD database being rated critical or high in severity, causing crashes, silent data corruption and arbitrary code execution. This vulnerability class persists in all kinds of C/C++ applications. While other types of memory corruption errors such as buffer overflows are nowadays harder to exploit due to mitigations, there are few mitigations deployed in production environments to prevent UAF vulnerabilities [43].

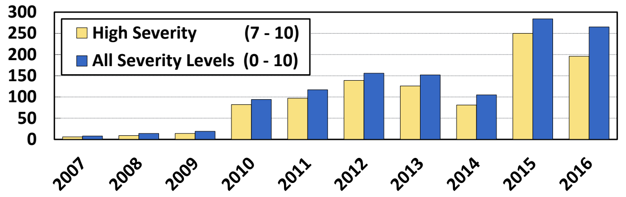


Figure 1: Use-after-free vulnerabilities in NVD [39].

There have been considerable efforts on building automatic tools for mitigating UAF bugs. However, existing solutions almost exclusively rely on dynamic analysis [7, 17, 22, 30, 41, 43], which inserts metadata-manipulating instrumentation code into the program, and detects or protects against UAF bugs at runtime by performing checks at all pointer dereferences [7, 22, 30, 41] or invalidating all dangling pointers identified [17, 43]. While maintaining zero or low false alarms (due to unsound modeling for, e.g., casting [22] and safety window sizes [7]), dynamic techniques have a number of limitations, including low code coverage (when used as debugging aids), binary incompatibility (due to memory layout transformations such as fat pointers [41]), and high runtime and memory overheads (due to runtime instrumentation).

Static analysis for detecting UAF bugs will not suffer from such instrumentation-based limitations. However, static techniques for UAF detection are scarce, although there are many for detecting other types of memory corruption bugs, such as buffer overflows [16, 19], memory leaks [8, 37] and null dereferences [10, 21].

In this paper, we introduce a new pointer-analysis-based static analysis for finding UAF bugs in multi-MLOC C programs efficiently and effectively. We first formulate the problem of detecting UAF bugs statically. We then describe several challenges faced, existing static techniques, and our solution (by highlighting its novelty).

Problem Statement. Consider a pair of statements, $\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big)$, where p and q are pointers and l_f and l_u are line numbers. Let $\mathcal{P}(l)$ be the set of all feasible (concrete) program paths reaching line l from `main()`. The pair is a UAF bug if and only if $\mathbb{ST}\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big)$ holds:

$$\boxed{\begin{array}{l}\textbf{[Spatio-Temporal Correlation]}\\ \mathbb{ST}\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big) \quad := \\ \exists\,(\rho_f, \rho_u) \in \mathcal{P}(l_f)\times\mathcal{P}(l_u) : (\rho_f, l_f) \rightsquigarrow (\rho_u, l_u) \,\wedge\, (\rho_f, p) \cong (\rho_u, q)\end{array}} \tag{1}$$

where $\rightsquigarrow$ denotes *temporal* reachability (in the program's ICFG (Interprocedural Control Flow Graph)) and $\cong$ denotes a *spatial* alias relation (meaning that p and q point to a common object). By convention, (ρ, l) identifies the program point l under a path abstraction ρ. Both temporal and spatial properties must correlate on the same concrete program path. However, $\mathbb{ST}$ is not computationally verifiable due to exponentially many paths in large codebases.

Challenges. One main challenge faced in designing a pointer-analysis-based static UAF analysis, $\mathcal{A}$, lies in how to reason about the exponential number of program paths in $\mathcal{P}(l_f) \times \mathcal{P}(l_u)$ in order to find real bugs at a low false positive rate. This entails approximating $\mathbb{ST}$ with $\mathbb{ST}^{\mathcal{A}}$ by abstracting these program paths with some contexts according to a tradeoff to be made among soundness, precision and scalability. $\mathcal{A}$ is *sound* (by catching all UAF bugs) if $\mathbb{ST}\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big) \Rightarrow \mathbb{ST}^{\mathcal{A}}\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big)$ for every UAF pair $\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big)$. $\mathcal{A}$ is *precise* (by reporting no false alarms if $\mathbb{ST}^{\mathcal{A}}\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big) \Rightarrow \mathbb{ST}\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big)$ for every $\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big)$. $\mathcal{A}$ is regarded as being *scalable* if $\mathbb{ST}^{\mathcal{A}}$ can analyze large codebases under a given budget. For convenience, $\mathbb{ST}^{\mathcal{A}}$ is also said to be sound/precise/scalable if $\mathcal{A}$ is sound/precise/scalable.

Another challenge is how to verify $\rightsquigarrow$ efficiently and precisely, especially in the presence of aliasing, as discussed below.

A final challenge lies in how to obtain $\cong$ efficiently and precisely. This requires a pointer analysis that is *field-sensitive* (by distinguishing different fields in a struct), *flow-sensitive* (by distinguishing flow of control), *context-sensitive* (by distinguishing calling contexts for a function), and *path-sensitive* (by distinguishing different program paths). However, computing such precise points-to information by reasoning about $\mathcal{P}(l_f) \times \mathcal{P}(l_u)$ is unscalable, despite recent advances on whole-program [12, 18, 42, 44] and demand-driven [13, 31, 35, 46] pointer analyses for C programs.

State of the Art. Due to the above challenges, there has been little work on developing specialized static approaches for detecting UAF bugs. General-purpose static approaches for detecting memory corruption bugs include model checking, abstract interpretation, pattern matching, and pointer analysis. Their corresponding representative tools are CBMC [14], Clang [3], Coccinelle [26], and Supa [35], which can be leveraged for finding UAF bugs.

Model Checking. CBMC [14] is a bounded model checker that reasons about all the program paths in $\mathcal{P}(l_f) \times \mathcal{P}(l_u)$ given in (1) for C/C++ programs as constraints that can be solved by an SMT solver. When used in finding UAF bugs, CBMC is sound (in a bounded manner) and highly precise but scales only to small programs [38] whose "sizes are restricted" (according to its user manual).

Abstract Interpretation. Clang [3] is an abstract interpreter for analyzing C/C++ programs. It adopts a highly unsound model by analyzing only a small subset of the program paths in $\mathcal{P}(l_f) \times \mathcal{P}(l_u)$ given in (1) in order to achieve scalability and precision. To scale for large codebases with few false alarms, Clang limits its UAF-bug-finding ability by performing an intraprocedural analysis (with inlining). In general, such tools refrain from reporting too many false alarms, but at the expense of missing many UAF bugs.

Pattern Matching. Coccinelle [26] is a pattern-based tool for analyzing and certifying C programs. Coccinelle can find UAF bugs based on some patterns given. Due to the lack of the points-to information, Coccinelle can be both fairly unsound and imprecise but is highly scalable (due to its pattern-matching nature).

Pointer Analysis. Supa [35] is a state-of-the-art demand-driven pointer analysis that is field-, flow- and context-sensitive but path-insensitive for C programs. When used in finding UAF bugs, Supa can be regarded as reasoning about all the program paths in $\mathcal{P}(l_f) \times \mathcal{P}(l_u)$ with an extremely coarse abstraction, $\{[\ ]\} \times \{[\ ]\}$, in order to achieve soundness and scalability. By convention, [ ] represents all possible calling contexts and thus all possible (concrete) paths reaching ℓ. Thus, $\mathbb{ST}$ in (1) is weakened significantly to $\mathbb{ST}^{\textsc{Supa}}$:

[Spatio-Temporal Correlation with a High Level of Spuriosity]

$$\mathbb{ST}^{\textsc{Supa}}\Big(\mathtt{free}(p@l_f), \mathtt{use}(q@l_u)\Big) \quad := \quad ([\ ], l_f) \rightsquigarrow ([\ ], l_u) \ \wedge\ ([\ ], p) \cong ([\ ], q) \tag{2}$$

where $\rightsquigarrow$ is the standard context-sensitive reachability and $\cong$ is the standard context-sensitive alias relation obtained under [ ].

When used in finding UAF bugs, $\mathbb{ST}^{\textsc{Supa}}$ will be highly imprecise, since spurious spatio-temporal correlations are introduced at an extremely large number of UAF pairs, where $\mathbb{ST}^{\textsc{Supa}}\Big(\mathtt{free}(p@l_f), \mathtt{use}(q@l_u)\Big) \not\Rightarrow \mathbb{ST}\Big(\mathtt{free}(p@l_f), \mathtt{use}(q@l_u)\Big)$ holds, as explained in Section 2 and validated in Section 5. These spurious correlations are false alarms.

Our Solution and Contributions. We introduce an (interprocedural) pointer-analysis-based static analysis, Stc, for finding UAF bugs in multi-MLOC C programs, by making several contributions.

First, we present a spatio-temporal context reduction technique that enables developing our new static UAF analysis systematically by simplifying $\mathbb{ST}$ in (1) into $\mathbb{ST}^{\textsc{Stc}}$ given below:

[Spatio-Temporal Context Reduction]

$$\mathbb{ST}^{\textsc{Stc}}\Big(\mathtt{free}(p@l_f), \mathtt{use}(q@l_u)\Big) \quad := \quad \exists\, (\widetilde{\rho_f}, \widetilde{\rho_u}) \in \widetilde{\mathcal{P}}(l_f) \times \widetilde{\mathcal{P}}(l_u) : (\widetilde{\rho_f}, l_f) \rightsquigarrow (\widetilde{\rho_u}, l_u) \ \wedge\ (\widetilde{\rho_f}, p) \cong (\widetilde{\rho_u}, q) \tag{3}$$

We ensure that $\mathbb{ST}^{\textsc{Stc}}$ is sound by requiring $\widetilde{\mathcal{P}}(l)$ to be a coarser abstraction of $\mathcal{P}(l)$ and scalable by requiring $|\widetilde{\mathcal{P}}(l_f) \times \widetilde{\mathcal{P}}(l_u)| \ll |\mathcal{P}(l_f) \times \mathcal{P}(l_u)|$. Unlike $\mathbb{ST}^{\textsc{Supa}}$, however, $\mathbb{ST}^{\textsc{Stc}}$ will be highly precise, as $\mathbb{ST}^{\textsc{Stc}}\Big(\mathtt{free}(p@l_f), \mathtt{use}(q@l_u)\Big) \not\Rightarrow \mathbb{ST}\Big(\mathtt{free}(p@l_f), \mathtt{use}(q@l_u)\Big)$ happens only for a small number of UAF pairs. *With spatio-temporal context reduction, Stc is designed purposely to preserve the spatio-temporal correlation of $\mathbb{ST}$ by keeping spurious correlations, i.e., false alarms, as low as possible.* Without it, Stc will be either highly unsound or highly imprecise.

Second, we adopt a multi-stage approach that starts with some UAF pairs obtained by a pre-analysis and then uses increasingly more precise yet more costly UAF analyses on increasingly fewer UAF pairs to filter out false alarms. In our current implementation, we perform context reduction by first using calling contexts and then considering path sensitivity. Staging such analyses this way improves the efficiency of the overall solution.

Third, we introduce a demand-driven pointer analysis with field-, flow-, context- and path-sensitivity as the foundation for the main analysis stages of Stc. This work is the first to consider path-sensitivity on-demand in order to reduce false UAF alarms.

Finally, we have implemented Stc in LLVM-3.8.0 and compared it with four different state-of-the-art static tools: CBMC (model checking) [14], Clang (abstract interpretation) [3], Coccinelle (pattern matching) [26], and Supa (pointer analysis) [35] using all the C test cases in Juliet Test Suite (JTS) [1] and 10 open-source C applications. For the ground truth evaluated with JTS, Stc is as effective as CBMC and Supa by detecting all the 138 known UAF bugs while Clang reports only 36 bugs and Coccinelle finds 126 bugs, with no false alarms issued in all the cases. For practicality evaluated with the 10 applications (totaling over 3 MLOC), Stc produces 132 warnings including 85 bugs in about 7.6 hours. In contrast, CBMC produces no warnings, terminating in 19.0 hours for the smallest application but exceeding the 3-day time budget for every remaining application; Clang reports 3 warnings including 1 bug in 1.2 hours; Coccinelle reports 103 false alarms in 179.0 seconds without finding any bugs; and Supa detects the same 85 bugs found by Stc, together with 23,095 false alarms, in 5.1 hours.

All the results can be reproduced at goo.gl/1HD8ZG.

2 OVERVIEW

As depicted in Figure 2, we start with a fast but imprecise "Pre-Analysis" (i.e., an Andersen-style pointer analysis [4]) to obtain a set of candidate UAF pairs to be analyzed (according to (1)). We then

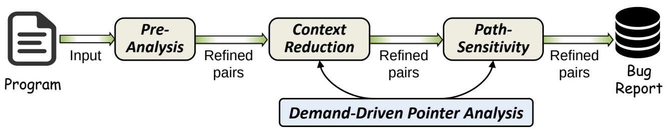


Figure 2: Workflow of Stc.

apply two spatio-temporal context reductions, "Calling Context Reduction" (Section 2.1) and "Path Reduction" (Section 2.2), founded on the same demand-driven pointer analysis infrastructure. Note that each stage refines the results from the preceding one.

We focus mostly on describing how calling-context reduction works and why it is significant. Without the two reduction techniques, a UAF analysis that relies on existing pointer analysis techniques will be either unscalable or highly imprecise (Section 5).

2.1 Calling-Context Reduction

The objective is to simplify $\mathbb{ST}$ in (1) into $\mathbb{ST}^{\text{Stc}}$ in (3) by abstracting program paths with calling contexts so that Stc is sound, scalable and highly precise. Our motivating example is given in Figure 3. We explain why Stc would be unscalable if full calling contexts were used (although it would be highly precise) and imprecise if k-limited calling contexts were used (although it would be possibly scalable). We achieve the best of both worlds by reducing full calling contexts substantially in both length and quantity.

2.1.1 Context-Sensitivity. We introduce the terminologies and notations used in context-sensitive program analysis.

- **Call String (or Call Stack).** In a *k-limited* or *k-callsite context-sensitive analysis*, every variable accessed or object allocated in a function *fun* is identified by a call string $c = [c_1, \dots, c_k]$, known as a *calling context*, which represents a sequence of the k-most-recent call sites (on the call stack) calling *fun*. In a call string, every recursion cycle is typically approximated once. The analysis is said to be *fully context-sensitive* if c_1 starts from `main()`.
- **Context-Sensitive Control-Flow Reachability.** Given two program points l and l' identified under contexts c and c', respectively, $(c, l) \rightsquigarrow (c', l')$ signifies that (c, l) *reaches context-sensitively* (c', l'). This is solved as a *balanced-parentheses problem* by matching calls and returns to filter out unrealizable paths in the program's ICFG [29]. We start from (c, l) with an abstract stack initialized as c. When entering a callee function from a callsite c_i, we push c_i into the context stack containing c, denoted $c \oplus [c_i]$. When returning from a callee to a callsite c_j, we pop c_j from the current stack containing c, denoted $c \ominus [c_j]$, if c contains c_j as its top value or $c = [\,]$ since a realizable path may start and end in different functions. Finally, $(c, l) \rightsquigarrow (c', l')$ is established if l' is reached when the context stack contains c'.
- **k-Call-Site Context-Sensitive Pointer Analysis.** Let $pt(c, v)$ be the points-to set of a variable v under a calling context c such that $|c| = k$. Given two variables p and q, $(c, p) \cong (c', q)$ holds if p and q may point to a common object, i.e., $pt(c, p) \cap pt(c', q) \neq \emptyset$. Here, c (c') represents the calling sequence for the function where p (q) is defined and h (h') represents the calling sequence for the function where object o is allocated. We speak of full context-sensitivity if c, c', h and h' all start from `main()`.

2.1.2 Limitations of k-Call-Site Context-Sensitivity. Figure 3(a) illustrates a typical heap usage scenario. In lines 1 – 11, there are 2^n calling contexts to `com()` from `main()`. In lines 12 – 35, two heap objects are allocated (lines 14 – 15), then used (lines 16 and 18), and finally, deallocated (lines 17 and 19), through a series of wrappers. There is one UAF pair $\big(\texttt{free(p@}ln34\texttt{)}, \texttt{use(q@}ln31\texttt{)}\big)$ to be analyzed, where `use(q@`$ln31$`)` stands for `print(*q)` at line 31.

This example is UAF-free. With full context-sensitivity, no warnings would be reported but the resulting analysis is unscalable. With k-limiting, the analysis scales, but at the expense of precision.

- **Full Context-Sensitivity: Precise but Unscalable.** As shown in Figure 3(b), $\boldsymbol{R}$ is the set of 2^n full calling contexts for `com()`. Thus, there are $2^{n+1} \times 2^{n+1}$ calling context pairs reaching $\big(\texttt{free(p)}, \texttt{print(*q)}\big)$. As $\forall c \in \boldsymbol{R}$: $(c \oplus [c_5, c_9], ln34) \rightsquigarrow (c \oplus [c_6, c_8], ln31)$, `free(p)` reaches `print(*q)`. However, as $\nexists\, c \in \boldsymbol{R}$, $([c \oplus [c_6, c_9], \texttt{p}) \cong ([c \oplus [c_6, c_8], \texttt{q})$, p and q never point to a common object. Therefore, no UAF warning will be issued. To reason about $\cong$, however, we may have to compute 2^{n+2} points-to sets shown in Figure 3(c), making both whole-program [12, 42] and demand-driven [33, 35] pointer analyses unscalable for large programs with a relatively large n (as validated later).
- **k-Limiting: Scalable but Imprecise.** With $k = 1$, the 2^{n+1} calling contexts reaching `free(p)` (`print(*q)`) are abstracted by $[c_9]$ ($[c_8]$). Then $([c_9], ln34) \rightsquigarrow ([c_8], ln31)$. In addition, $([c_9], \texttt{p}) \cong ([c_8], \texttt{q})$ holds spuriously, based on the two points-to sets, as shown in Figure 3(d), computed efficiently but imprecisely. As a result, a false alarm (from line 17 to line 18) is reported.

  With 2-limiting, the false alarm will be suppressed. However, increasing k will not work for large codebases for two reasons. First, the number of context pairs to be analyzed at a UAF pair will grow exponentially. Second, the optimal values for k vary across the UAF pairs. Finding such values is beyond the state-of-the art.

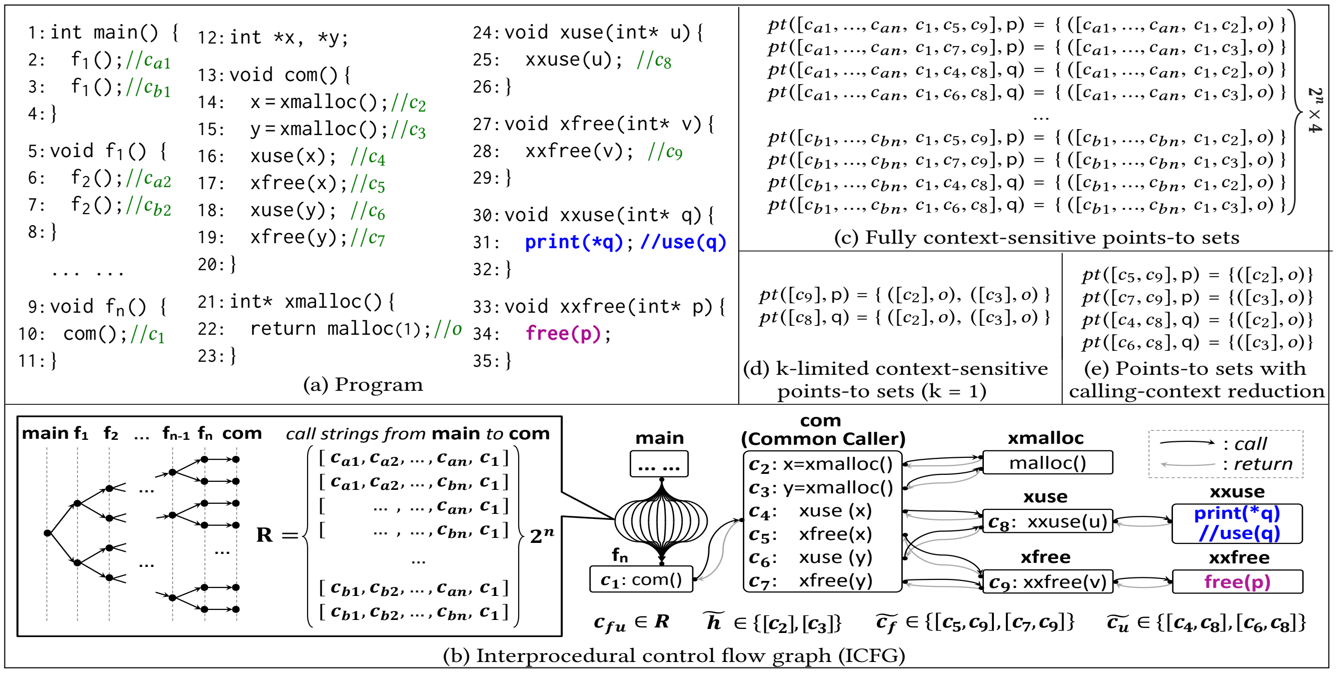
2.1.3 Spatio-Temporal Calling-Context Reduction. The key insight is to remove prefixes in full calling contexts that do not contribute to context-sensitivity, thereby achieving the precision of full context-sensitivity and the scalability of k-limiting.

Let $pt^{\infty}(c, v)$ be the points-to set of v under context c computed by an oracle pointer analysis fully context-sensitively. As illustrated in Figure 4, $\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big)$ is a bug when C1 – C4 hold:

(C1): `main()` calls, under a context c_{fu}, a *common caller function*, which calls an object allocation function, e.g., `malloc()`, `free`(p) and `use`(q) at lines l_o, l_p and l_2 in that order,
(C2): o is allocated under context $c_{fu} \oplus \widetilde{h}$,
(C3): $(c_{fu} \oplus \widetilde{h}, o) \in pt^{\infty}(c_{fu} \oplus \widetilde{c_f}, p)$, and
(C4): $(c_{fu} \oplus \widetilde{h}, o) \in pt^{\infty}(c_{fu} \oplus \widetilde{c_u}, q)$.

By definition, $(c_f, l_f) \rightsquigarrow (c_u, l_u) \wedge (c_f, p) \cong (c_u, q) \iff (\widetilde{c_f}, l_f) \rightsquigarrow (\widetilde{c_u}, l_u) \wedge (\widetilde{c_f}, p) \cong (\widetilde{c_u}, q)$, making c_{fu} redundant.

For our example, Figure 3(b) illustrates the calling context reduction performed. As $c_{fu} \in \boldsymbol{R}$ is a common prefix for the common caller, `com()`, that satisfies C1 – C4, a total of $2^{n+1} \times 2^{n+1}$ full calling context pairs reaching $\big(\texttt{free(p)}, \texttt{print(*q)}\big)$ have been reduced to just four, with $(\widetilde{c_f}, \widetilde{c_u}) \in \{[c_5, c_9], [c_7, c_9]\} \times \{[c_4, c_8], [c_6, c_8]\}$ and $\widetilde{h} \in \{c_2, c_3\}$. As `com()` is a common caller, $(c \oplus \widetilde{c_f}, \texttt{p}) \cong (c' \oplus \widetilde{c_u}, \texttt{q})$



```
1: int main() {
2:   f1(); //c_a1
3:   f1(); //c_b1
4: }
5: void f1() {
6:   f2(); //c_a2
7:   f2(); //c_b2
8: }
   ... ...
9: void fn() {
10:  com(); //c1
11: }
```

```
12: int *x, *y;
13: void com() {
14:   x = xmalloc(); //c2
15:   y = xmalloc(); //c3
16:   xuse(x); //c4
17:   xfree(x); //c5
18:   xuse(y); //c6
19:   xfree(y); //c7
20: }
21: int* xmalloc() {
22:   return malloc(1); //o
23: }
```

```
24: void xuse(int* u) {
25:   xxuse(u); //c8
26: }
27: void xfree(int* v) {
28:   xxfree(v); //c9
29: }
30: void xxuse(int* q) {
31:   print(*q); //use(q)
32: }
33: void xxfree(int* p) {
34:   free(p);
35: }
```

(a) Program

$$\left.\begin{array}{l}
pt([c_{a1},\dots,c_{an},\ c_1,c_5,c_9],\mathrm{p}) = \{\,([c_{a1},\dots,c_{an},\ c_1,c_2],o)\,\}\\
pt([c_{a1},\dots,c_{an},\ c_1,c_7,c_9],\mathrm{p}) = \{\,([c_{a1},\dots,c_{an},\ c_1,c_3],o)\,\}\\
pt([c_{a1},\dots,c_{an},\ c_1,c_4,c_8],\mathrm{q}) = \{\,([c_{a1},\dots,c_{an},\ c_1,c_2],o)\,\}\\
pt([c_{a1},\dots,c_{an},\ c_1,c_6,c_8],\mathrm{q}) = \{\,([c_{a1},\dots,c_{an},\ c_1,c_3],o)\,\}\\
\cdots\\
pt([c_{b1},\dots,c_{bn},\ c_1,c_5,c_9],\mathrm{p}) = \{\,([c_{b1},\dots,c_{bn},\ c_1,c_2],o)\,\}\\
pt([c_{b1},\dots,c_{bn},\ c_1,c_7,c_9],\mathrm{p}) = \{\,([c_{b1},\dots,c_{bn},\ c_1,c_3],o)\,\}\\
pt([c_{b1},\dots,c_{bn},\ c_1,c_4,c_8],\mathrm{q}) = \{\,([c_{b1},\dots,c_{bn},\ c_1,c_2],o)\,\}\\
pt([c_{b1},\dots,c_{bn},\ c_1,c_6,c_8],\mathrm{q}) = \{\,([c_{b1},\dots,c_{bn},\ c_1,c_3],o)\,\}
\end{array}\right\}2^n\times 4$$

(c) Fully context-sensitive points-to sets

$$pt([c_9],\mathrm{p}) = \{\,([c_2],o),\ ([c_3],o)\,\}$$
$$pt([c_8],\mathrm{q}) = \{\,([c_2],o),\ ([c_3],o)\,\}$$

(d) k-limited context-sensitive points-to sets (k = 1)

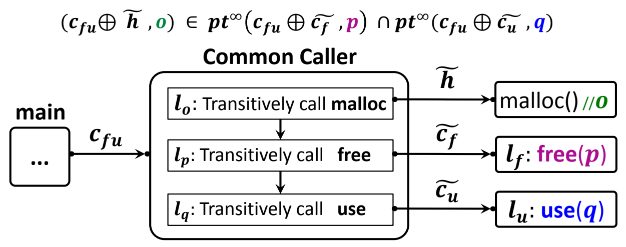
$$pt([c_5,c_9],\mathrm{p}) = \{([c_2],o)\}$$
$$pt([c_7,c_9],\mathrm{p}) = \{([c_3],o)\}$$
$$pt([c_4,c_8],\mathrm{q}) = \{([c_2],o)\}$$
$$pt([c_6,c_8],\mathrm{q}) = \{([c_3],o)\}$$

(e) Points-to sets with calling-context reduction

main f1 f2 ... fn-1 fn com
call strings from main to com
$$\mathbf{R} = \left\{\begin{array}{l}[\,c_{a1}, c_{a2}, \dots, c_{an}, c_1\,]\\ [\,c_{a1}, c_{a2}, \dots, c_{bn}, c_1\,]\\ [\quad \dots, \dots, c_{an}, c_1\,]\\ [\quad \dots, \dots, c_{bn}, c_1\,]\\ \dots\\ [\,c_{b1}, c_{b2}, \dots, c_{an}, c_1\,]\\ [\,c_{b1}, c_{b2}, \dots, c_{bn}, c_1\,]\end{array}\right\} 2^n$$
main
... ...
fn
c1: com()
com
(Common Caller)
c2: x=xmalloc()
c3: y=xmalloc()
c4: xuse (x)
c5: xfree(x)
c6: xuse (y)
c7: xfree(y)
xmalloc
malloc()
xuse
c8: xxuse(u)
xfree
c9: xxfree(v)
xxuse
print(*q) //use(q)
xxfree
free(p)
: call
: return
$c_{fu} \in \mathbf{R}$
$\widetilde{h} \in \{[c_2],[c_3]\}$
$\widetilde{c_f} \in \{[c_5,c_9],[c_7,c_9]\}$
$\widetilde{c_u} \in \{[c_4,c_8],[c_6,c_8]\}$

(b) Interprocedural control flow graph (ICFG)

Figure 3: Calling-context reduction for overcoming the limitations of fully and k-limited context sensitivity in UAF detection.



$$(c_{fu} \oplus \widetilde{h}, o) \in pt^{\infty}(c_{fu} \oplus \widetilde{c_f}, p) \cap pt^{\infty}(c_{fu} \oplus \widetilde{c_u}, q)$$

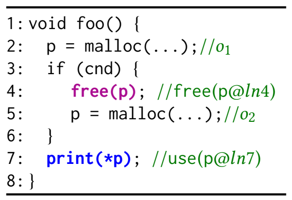
Figure 4: Context reduction, illustrated conceptually with an oracle fully-context-sensitive pointer analysis.

does not hold, i.e., p and q are must-not-aliases if c and c' are different prefixes in $\boldsymbol{R}$. Thus, it is only necessary to verify $(c \oplus \widetilde{c_f}, ln34) \rightsquigarrow (c' \oplus \widetilde{c_u}, ln31)$ when $c = c'$. We can do this efficiently by checking if $car(\widetilde{c_f})$ appears lexically before $car(\widetilde{c_u})$ in com(), i.e., if l_p appears before l_q in Figure 4. Note that *car* is the standard function for returning the first element in a sequence. For the four reduced context pairs, only $([c_5, c_9], ln34) \rightsquigarrow ([c_6, c_8], ln31)$ holds since c_5 precedes c_6 in com(). According to the points-to sets, shown in Figure 3(e), computed efficiently for the reduced calling contexts, $([c_5, c_9], \mathrm{p}) \not\cong ([c_6, c_8], \mathrm{q})$. Hence, no UAF warnings are reported.

In Section 3, we will give an inference rule for performing the calling context reduction illustrated above.

2.2 Path Reduction

We improve precision by augmenting calling contexts with path-sensitivity. Consider a bug-free example in Figure 5. Without path-sensitivity, p at line 4 points to o_1 and q at line 7 points to o_1 and o_2, causing a path-insensitive detector to report a false alarm



```
1: void foo() {
2:   p = malloc(...); //o1
3:   if (cnd) {
4:     free(p); //free(p@ln4)
5:     p = malloc(...); //o2
6:   }
7:   print(*p); //use(p@ln7)
8: }
```

Figure 5: Path reduction.

$\big($free(p@$ln4$), use(p@$ln7$)$\big)$. With path-sensitivity, however, this false alarm will be suppressed successfully.

In Section 3, we will give an inference rule for path reduction. To the best of our knowledge, Stc is the first UAF detector for large codebases that reasons about path-sensitivity on-demand based on a new path-sensitive demand-driven pointer analysis.

3 THE STC ANALYSIS FOR UAF DETECTION

As shown in Figure 2, Stc comprises three key components: ① spatio-temporal context reduction, ② demand-driven pointer analysis, and ③ multi-stage UAF analysis. While ① represents the most important contribution of this paper, we introduce ② and ③ first in that order in order to build the basis for ①.

3.1 Demand-Driven Pointer Analysis

We describe a demand-driven pointer analysis that is not only field-, flow- and context-sensitive as in [33, 35] but also path-sensitive. Adding path-sensitivity is significant in terms of both advancing demand-driven pointer analysis in general and reducing a large number of false alarms that would otherwise be reported by Stc.

3.1.1 Program Representation. A C program is represented by putting it into LLVM's partial SSA form, following [12, 18, 20, 42]. The set of program variables $\mathcal{V}$ is separated into two subsets: $\mathcal{A}$ containing all possible targets, i.e., *address-taken variables* of a pointer, and $\mathcal{T}$ containing all *top-level variables*, where $\mathcal{V} = \mathcal{T} \cup \mathcal{A}$.

After the SSA conversion, a program has seven types of statements: $p = \&a$ (AddrOf), $p = q$ (Copy), $p = *q$ (Load), $*p = q$ (Store), $p = \phi(\dots, q, \dots)$ (Phi), $p = \texttt{call}\, fun(q)$ (Call), and $\texttt{return}\, p$ (Return), where $p, q \in \mathcal{T}$ and $a \in \mathcal{A}$. Top-level variables are put directly in SSA form while address-taken variables are accessed indirectly via Load or Store. For an AddrOf statement $p = \&a$, known as an *allocation site*, a is a stack or global variable with its address taken or a dynamically created abstract heap object. Passing parameters and return values (explicitly for top-level and implicitly for address-taken variables) is modeled by Copy.

All pointer analyses used are field-sensitive. Each field instance of a struct is treated as a separate object. However, arrays are considered monolithic. Precise solutions for arrays do not exist.

Given a program, its ICFG is built in the normal manner [15]. A call site for a function *fun* is split into a call node and a return node, with a call edge from the call node to the entry node of *fun* and a return edge from the exit node of *fun* to the return node.

$$\texttt{[ADDR]}\ \frac{c,\tau,l:p=\&o}{(c,\tau,l,p)\hookleftarrow(c,\tau,\widehat{o})}$$

$$\texttt{[COPY]}\ \frac{c,\tau,l:p=q \qquad l_q\overset{q}{\to}l \qquad \delta_q=Guard(l_q,l)}{(c,\tau,l,p)\hookleftarrow(c,\tau\wedge\delta_q,l_q,q)}$$

$$\texttt{[PHI]}\ \frac{c,\tau,l:p=\phi(..,q,..) \qquad l_q\overset{q}{\to}l \qquad \delta_q=Guard(l_q,l)}{(c,\tau,l,p)\hookleftarrow(c,\tau\wedge\delta_q,l_q,q)}$$

$$\texttt{[LOAD]}\ \frac{\begin{array}{cc} c,\tau,l:p=*q & (c,\tau\wedge\delta_q,l_q,q)\hookleftarrow(c_o,\tau_o,\widehat{o}) \\ l_q\overset{q}{\to}l \quad l_o\overset{o}{\to}l & \delta_q=Guard(l_q,l) \quad \delta_o=Guard(l_o,l)\end{array}}{(c,\tau,l,p)\hookleftarrow(c_o,\tau_o\wedge\delta_o,l_o,o)}$$

$$\texttt{[Store]}\ \frac{\begin{array}{ccc} c,\tau,l:*p=q & (c,\tau\wedge\delta_p,l_p,p)\hookleftarrow(c_o,\tau_o,\widehat{o}) & \\ l_p\overset{p}{\to}l & l_q\overset{q}{\to}l & l_o\overset{o}{\to}l \\ \delta_p=Guard(l_p,l) & \delta_q=Guard(l_q,l) & \delta_o=Guard(l_o,l)\end{array}}{\begin{array}{c}(c_o,\tau_o\wedge\delta_o,l,o)\hookleftarrow(c,\tau,l_q,q)\\(c_o,\tau_o,l,o)\hookleftarrow(c_o,\tau_o\wedge\delta_o,l_o,o)\end{array}}$$

$$\texttt{[Call]}\ \frac{\begin{array}{cc} c,\tau,l:\textbf{define}\ fun(v)\ \{...\} & l_{call}:\textbf{call}\ fun(a) \\ l_a\overset{a}{\to}l_{call} & \delta_a=Guard(l_a,l_{call})\end{array}}{(c,\tau,l,v)\hookleftarrow(c\ominus[l],\tau\wedge\delta_a,l_a,a)}$$

$$\texttt{[Return]}\ \frac{\begin{array}{cc} c,\tau,l:y=\textbf{call}\ fun(...) & \textbf{define}\ fun(...)\ \{...,\ l_{ret}:\textbf{return}\ x\} \\ l_x\overset{x}{\to}l_{ret} & \delta_x=Guard(l_x,l_{ret})\end{array}}{(c,\tau,l,y)\hookleftarrow(c\oplus[l],\tau\wedge\delta_x,l_x,x)}$$

$$\texttt{[Trans]}\ \frac{(c,\tau,l,v)\hookleftarrow(c',\tau',l',v') \qquad (c',\tau',l',v')\hookleftarrow(c'',\tau'',l'',v'')}{(c,\tau,l,v)\hookleftarrow(c'',\tau'',l'',v'')}$$

Figure 6: Demand-driven pointer analysis with field-, flow- and context-sensitivity as in [35] and path-sensitivity added.

3.1.2 Algorithm. As shown in Figure 6, we extend [33, 35] by making it also path-sensitive with the required path guards generated on-demand. Our analysis is flow-sensitive, since it answers a points-to query for a variable v by traversing all the def-use chains affecting v backwards on a value-flow graph (VFG) [12, 36, 37]. In the VFG, a node represents a statement (identified by its line number) and an edge from statement l to statement l', denoted $l\overset{v}{\to}l'$, represents a def-use relation for a variable $v\in\mathcal{V}$, with its def at statement l and its use at statement l'. These def-use chains are pre-computed with a fast but imprecise Andersen-style pointer analysis flow- and context-insensitively [4]. Our analysis is also context-sensitive. The points-to query $pt([c_1,\ldots,c_k],v)$, where c_i identifies a call site, returns the points-to set of v for all the function calling sequences ending with $[c_1,\ldots,c_k]$. Thus, $pt([\,],v@l)$ gives the points-to set of v at line l at all calling contexts.

We explain our extension on handling path-sensitivity highlighted in red. Calling contexts are path abstractions but can be too coarse. To perform path-sensitive analysis, we represent an abstract path by both a calling context c and a path guard τ so that c specifies its calling sequence and τ collects its branch conditions. Thus, $pt((c,\tau),v@l)$ gives the points-to set of v at line l under (c,τ).

In a function *fun*, every branch condition is treated as a Boolean formula. As in [8, 37], a loop (after unrolling, if needed) is approximated only once with its backedge ignored. For each control-flow edge e, $EdgeGuard(e)$ is the branch condition under which e is executed. For a control-flow path cp, which consists of a set of control-flow edges e, the path condition is the logical conjunction of branch conditions of e, i.e., $\bigwedge_{e\in cp}EdgeGuard(e)$. A *path guard* $Guard(l,l')$ from a statement l to a statement l' in *fun* is the logical disjunction of path condition of all control-flow paths from l to l':

$$\boxed{Guard(l,l') \;=\; \bigvee_{cp\in Path(l,l')}\ \bigwedge_{e\in cp} EdgeGuard(e)} \qquad (4)$$

where $Path(l,l')$ denotes the set of control-flow paths from l to l'.

A *path guard* τ from the entry of `main()` to a statement is defined simply in terms of (4). For the two special cases, true (false) represents an abstract feasible (infeasible) path.

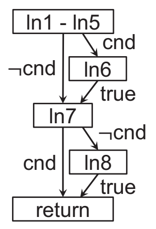
Given (c,τ,l,v), where variable v appears at line l, the points-to set of v is computed by finding all reachable objects $(c_o,\tau_o,\widehat{o})$ via backward traversal on the pre-computed def-use chains:

$$\boxed{pt((c,\tau),v@l)=\{(c_o,\tau_o,o)\mid(c,\tau,l,v)\hookleftarrow(c_o,\tau_o,\widehat{o})\}} \qquad (5)$$

The first seven rules handle the seven types of statements in the program by traversing backwards along all the pre-computed def-use chains affecting $v@l$. The last says that $\hookleftarrow$ is transitive. In `[CALL]`, $a\in\mathcal{V}$ denotes a variable passed into the callee directly or indirectly via parameter passing. Similarly, x in `[Return]` represents a value returned directly or indirectly from the callee to its caller. Context-sensitivity is enforced by matching calls and returns. In $c\oplus[l]$, the callsite label l is appended to c. In $c\ominus[l]$, l is removed from c if c contains l as its top value or is empty since a realizable path may start and end in different functions [34]. Strong updates are performed on singleton objects as in Supa [35].

For a program, its call graph is built on the fly. Our analysis handles its SCCs (Strongly Connected Components) context-sensitively but the function calls in a SCC context-insensitively as in [34]. Thus, our analysis is fully field- and flow-sensitive as well as fully context- and path-sensitive (modulo loops and recursion cycles).

```
   main() {
1:  p = &o1;
2:  x = &o2;
3:  *p = x;
4:  y = &o3
5:  if(cnd)
6:    *p = y;
7:  if(!cnd)
8:    z = *p;
   }
```
(a) Code



(b) CFG

$Guard(ln1,ln2) =$ true
$Guard(ln1,ln3) =$ true
$Guard(ln1,ln6) =$ cnd

$Guard(ln1,ln8) = Guard(ln3,ln8)$
$= (\text{cnd}\wedge\neg\text{cnd})\vee(\neg\text{cnd}\wedge\neg\text{cnd})$
$= \neg\text{cnd}$

$Guard(ln6,ln8) = \neg\text{cnd}$

(c) Path Guards

Figure 7: Path guard construction on a CFG.

3.1.3 Example. We use an example in Figure 7 to explain our rules on on-demand path-guard generation, with some relevant path guards shown. Suppose a points-to query $pt(([\,],\text{true}),$ `z@ln8`$)$ is issued. With path-sensitivity, we can determine precisely that z

points only to o_2 but not o_3. In line 8, [Load] is applied:

$$\frac{[\,], \mathsf{true}, ln8 : \mathtt{z = *p} \qquad ([\,], \neg\mathsf{cnd}, ln1, \mathsf{p}) \hookleftarrow ([\,], \neg\mathsf{cnd}, \widehat{o_1}) \qquad ln1 \xrightarrow{\mathsf{p}} ln8 \quad \delta_{\mathsf{p}} = Guard(ln1, ln8) = \neg\mathsf{cnd} \qquad \delta_{o_1} = Guard(ln3, ln8) = \neg\mathsf{cnd} \qquad ln3 \xrightarrow{o_1} ln8}{([\,], \mathsf{true}, ln8, \mathsf{z}) \hookleftarrow ([\,], \neg\mathsf{cnd}, ln3, o_1)}$$

Similarly, applying [STORE] and [ADDR] to lines 3 and 2, respectively, yields $([],\mathsf{true}, ln8, \mathsf{z}) \hookleftarrow ([],\neg\mathsf{cnd}, ln3, o_1) \hookleftarrow ([],\neg\mathsf{cnd}, ln2, \mathsf{x}) \hookleftarrow ([],\neg cnd, \widehat{o_2})$. Thus, z points to o_2. We can also attempt to trace z backwards to o_3 via `*p = y`, by first applying [Load] in line 8, which produces $([\,], true, ln8, \mathsf{z}) \hookleftarrow ([\,], \neg\mathsf{cnd}, ln6, o_1)$. However, no more rules can be applied further, because $([\,], \neg\mathsf{cnd}, ln6, o_1) \not\hookleftarrow ([\,], \neg\mathsf{cnd} \wedge \mathsf{cnd}, ln6, \mathsf{y})$, as $\neg\mathsf{cnd} \wedge \mathsf{cnd} = \mathsf{false}$, representing an infeasible path. Thus, z cannot point to o_3.

3.2 Multi-Stage UAF Analysis

Stc, as shown in Figure 2, consists of two stages, Stages 1 and 2. Each stage decides whether to issue a warning or not for a given UAF pair by verifying its own version of $\mathbb{ST}^{\textsc{Stc}}$ given in (3), which is discussed below. The pre-analysis, which serves to provide the set of UAF pairs for Stc to analyze, can be regarded as Stage 0.

Each stage is founded on a pointer analysis, P_i. In Stage 0 (our pre-analysis), P_0 is flow-, context- and path-insensitive. In Stage 1 (with calling-context reduction), P_1 is flow- and context-sensitive on-demand. In Stage 2 (with path reduction), P_2 is also path-sensitive. As i increases, Stage i becomes progressively more precise but also more costly, working on filtering out false alarms from increasingly a smaller set of UAF warnings provided by Stage $i-1$.

At Stage i, where $1 \leqslant i \leqslant 2$, we obtain $\rightsquigarrow$ and $\cong$ as follows. To obtain $\cong$, we invoke P_i to compute the points-to set $pt_i(\rho_i, v)$, with pt in (5) subscripted by i, for every variable v needed on-demand under a budget η_i. Here, ρ_i is an appropriate path abstraction used by P_i for querying v. If η_i is exhausted before $pt_i(\rho_i, v)$ is found, we fall back to P_{i-1} by setting $pt_i(\rho_i, v) = pt_{i-1}(\rho_{i-1}, v)$ conservatively, where the set of concrete paths abstracted by ρ_i is a subset of the set of concrete paths abstracted by ρ_{i-1}. To obtain $\rightsquigarrow$, we compute it on the ICFG obtained in Stage 0 and refined with the function pointers being resolved more precisely by P_i.

3.3 Spatio-Temporal Context Reduction

We describe two reductions performed for Stages 1 and 2, with the latter being developed on top of the former, making Stage 2 more precise but also more costly than Stage 1. For each stage, we give the inference rules for implementing for its reduction.

3.3.1 Stage 1. Calling-Context Reduction.

We abstract program paths with calling contexts so that the resulting UAF analysis is sound, scalable and highly precise (with as few spurious correlations as possible), as already motivated in Section 2. To this end, we would like to replace $\mathcal{P}(l_f) \times \mathcal{P}(l_u)$ in (1) with a coarser abstraction $\widetilde{C}(l_f) \times \widetilde{C}(l_u)$ expressed in terms of calling contexts, reduced as shown in Figure 4, so that $\mathbb{ST}$ in (1) can simplify to $\mathbb{ST}^C$:

[Spatio-Temporal Calling Context Reduction]

$$\mathbb{ST}^C\Big(\mathtt{free}(p@l_f), \mathtt{use}(q@l_u)\Big) \quad := \quad \exists\, (\widetilde{c_f}, \widetilde{c_u}) \in \widetilde{C}(l_f) \times \widetilde{C}(l_u) : (\widetilde{c_f}, l_f) \rightsquigarrow (\widetilde{c_u}, l_u) \ \wedge\ (\widetilde{c_f}, p) \cong (\widetilde{c_u}, q) \qquad (6)$$

How do we construct $\widetilde{C}(l_f) \times \widetilde{C}(l_u)$? The basic idea is illustrated conceptually in Figure 4 with an oracle fully-context-sensitive pointer analysis, pt^∞. To reduce the number of context pairs in $\widetilde{C}(l_f) \times \widetilde{C}(l_u)$, we should remove their redundant prefixes if they do not help separate calling contexts as desired.

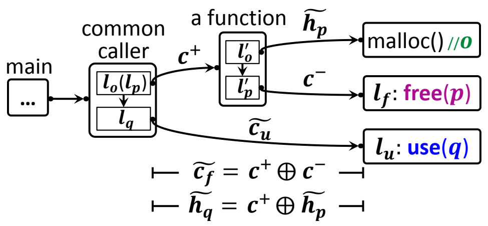


Figure 8: Context reduction with a demand-driven flow- and context-sensitive pointer analysis that computes the points-to set of a variable under the calling context [ ]. Boxes and arrows represent functions and (transitive) calls, respectively.

However, pt^∞ is non-existent as it is not scalable for reasonably large programs. Below we obtain $\widetilde{C}(l_f) \times \widetilde{C}(l_u)$ equivalently by using pt_1, which is a flow- and context-sensitive pointer analysis in Stage 1 (Section 3.2), with the intuition illustrated in Figure 8:

$$[\texttt{CTX-R}]\ \frac{\begin{array}{c}(\widetilde{h_p}, o) \in pt_1([\,], p) \quad (\widetilde{h_q}, o) \in pt_1([\,], q) \quad \widetilde{h_p} \text{ is a suffix of } \widetilde{h_q} \\ l_o = car(\widetilde{h_q} \oplus [o]) \qquad l_p = car(\widetilde{c_f} \oplus [l_f]) \qquad l_q = car(\widetilde{c_u} \oplus [l_u]) \\ l_p \text{ and } l_q \text{ reside in the function containing } l_o \\ \widetilde{c_f} \text{ is a calling context for } l_f \qquad \widetilde{c_u} \text{ is a calling context for } l_u\end{array}}{(\widetilde{c_f}, \widetilde{c_u}) \in \widetilde{C}(l_f) \times \widetilde{C}(l_u)} \qquad (7)$$

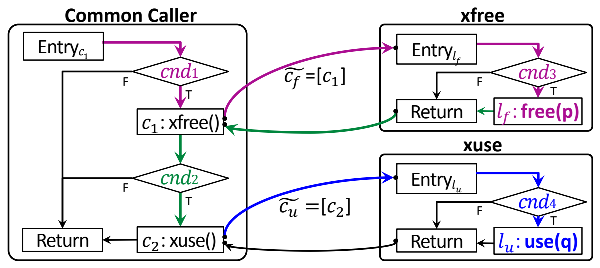
Figure 8 illustrates the generic scenario in which $\Big(\mathtt{free}(p@l_f), \mathtt{use}(q@l_u)\Big)$ may be potentially a UAF bug. In the special case when the function that contains lines l'_o and l'_p is the same as the common caller that contains line l_q, we will be back to the scenario illustrated earlier in Figure 4.

As shown in Figure 6, $pt([\,], v)$ is computed on-demand by traversing interprocedurally only the statements producing values that may flow into v, under all possible calling contexts for the function, fun, that contains v, as indicated by [ ]. If v is found to point to o when the calling sequence (i.e., calling context) is h (from fun to o's allocation site), then $(h, o) \in pt([\,], v)$.

Let us examine [CTX-R]. As $(\widetilde{h_p}, o) \in pt_1([\,], p)$ and $(\widetilde{h_q}, o) \in pt_1([\,], q)$, $(c_f, p) \cong (c_u, q)$ holds if $\widetilde{h_p}$ is a suffix of $\widetilde{h_q}$, in which case, $(\widetilde{h_p}, o)$ and $(\widetilde{h_q}, o)$ may represent a common (concrete) object. The common caller, shown in Figure 4, is the function that contains l_o. Thus, $\widetilde{c_f}$ is simply the context from a callsite $car(\widetilde{c_f} \oplus [l_f])$ in the same caller function to l_f. Similarly, $\widetilde{c_u}$ is derived.

Let $\mathbb{ST}^\infty$ be obtained from $\mathbb{ST}^C$ such that $\widetilde{C}(l_f) \times \widetilde{C}(l_u)$ are now expressed in terms of all full calling contexts possible. By [CTX-R], $\mathbb{ST}^C\Big(\mathtt{free}(p@l_f), \mathtt{use}(q@l_u)\Big) \iff \mathbb{ST}^\infty\Big(\mathtt{free}(p@l_f), \mathtt{use}(q@l_u)\Big)$. Thus, $\mathbb{ST}^C$ is sound and as precise as possible by using calling contexts. In addition, $\mathbb{ST}^C$ is efficiently verifiable, as motivated in Section 3 and validated later.

By construction, $(c_{fu} \oplus \widetilde{c_f}, p) \ncong (c'_{fu} \oplus \widetilde{c_u}, q)$, i.e., p and q are must-not-aliases if c_{fu} and c'_{fu} are different context prefixes. Now, $(c_{fu} \oplus \widetilde{c_f}, l_f) \rightsquigarrow (c'_{fu} \oplus \widetilde{c_u}, l_u)$ holds, where $c_{fu} = c'_{fu}$, i.e., $(\widetilde{c_f}, l_f) \rightsquigarrow (\widetilde{c_u}, l_u)$ holds, only if l_p appears lexically before l_q in the common caller containing l_0 in [CTX-R]. To check $(\widetilde{c_f}, p) \cong (\widetilde{c_u}, q)$ for these reachable pairs, we rely on $pt_1(\widetilde{c_f}, p)$ and $pt_1(\widetilde{c_u}, q)$.



$$\widetilde{\tau_f} \;=\; \mathit{Guard}(\text{Entry}_{c_1}, c_1) \wedge \mathit{Guard}(\text{Entry}_{l_f}, l_f) = \; cnd_1 \wedge cnd_3$$

$$\widetilde{\tau_u} \;=\; \underline{\widetilde{\tau_f}} \wedge \underline{\mathit{Guard}(l_f, c_2)} \wedge \underline{\mathit{Guard}(\text{Entry}_{l_u}, l_u)} = \; cnd_1 \wedge cnd_3 \wedge cnd_2 \wedge cnd_4$$

Figure 9: Adding path guards to calling contexts in `[PAT-R]`.

Let us apply `[Ctx-R]` to formally analyze the UAF pair $\big($`free`$(\texttt{p@}ln34)$, `use`$(\texttt{q@}ln31)\big)$ in Figure 3. By computing on-demand the points-to sets of p and q flow- and context-sensitively, we obtain $pt_1([\,],\texttt{p}) = pt_1([\,],\texttt{q}) = \{(c_2, o), (c_3, o)\}$. Let us consider $(c2, o)$ only. For this example, considering also $(c3, o)$ adds no information. As $\widetilde{h_p} = \widetilde{h_q} = [c_2]$, we have $l_o = car([c2, o]) = c_2$. Thus, `com()` is the common caller that transitively calls `malloc()`, `free(p)` and `use(q)`. As $l_p \in \{c_5, c_7\}$ and $l_q \in \{c_4, c_6\}$, we obtain $\widetilde{C}(l_f) \times \widetilde{C}(l_u) = \{[c_5, c_9], [c_7, c_9]\} \times \{[c_4, c_8], [c_6, c_8]\}$. Finally, the UAF pair is filtered out as a false alarm, as discussed in Section 2.1.

3.3.2 Stage 2. Path Reduction. We improve calling-context reduction by augmenting the calling contexts $\widetilde{c} \in \widetilde{C}(l)$ from Stage 1 with path guards $\widetilde{\tau} \in \widetilde{\mathcal{G}(l)}$, thus achieving path reduction. As a result, $\mathbb{ST}^C$ is refined to $\mathbb{ST}^P$ by considering path-sensitivity:

[Spatio-Temporal Path Reduction]

$$\begin{aligned} &\mathbb{ST}^P\big(\texttt{free}(p@l_f), \texttt{use}(q@l_u)\big) \quad := \\ &\exists\, \big((\widetilde{c_f}, \widetilde{\tau_f}), (\widetilde{c_u}, \widetilde{\tau_u})\big) \in \Big(\big(\widetilde{C}(l_f) \times \widetilde{\mathcal{G}}(l_f)\big) \times \big(\widetilde{C}(l_u) \times \widetilde{\mathcal{G}}(l_u)\big)\Big) : \\ &\quad ((\widetilde{c_f}, \widetilde{\tau_f}), l_f) \rightsquigarrow ((\widetilde{c_u}, \widetilde{\tau_u}), l_u) \;\wedge\; ((\widetilde{c_f}, \widetilde{\tau_f}), p) \cong ((\widetilde{c_u}, \widetilde{\tau_u}), q) \end{aligned} \tag{8}$$

where $\big(\widetilde{C}(l_f) \times \widetilde{\mathcal{G}}(l_f)\big) \times \big(\widetilde{C}(l_u) \times \widetilde{\mathcal{G}}(l_u)\big)$ is constructed below:

$$\texttt{[PAT-R]}\;\frac{\begin{array}{c} \widetilde{c_f} \in \widetilde{C}(l_f) \qquad \widetilde{c_u} \in \widetilde{C}(l_u) \qquad \widetilde{\tau_f} = \bigwedge\limits_{c_i \in \widetilde{c_f} \oplus [l_f]} \mathit{Guard}(\text{ENTRY}_{c_i}, c_i) \\ \widetilde{\tau_u} = \underline{\widetilde{\tau_f}} \wedge \underline{\mathit{Guard}(l_f, \mathit{car}(\widetilde{c_u} \oplus [l_u]))} \wedge \underline{\Big(\bigwedge\limits_{c_i \in \mathit{cdr}(\widetilde{c_u} \oplus [l_u])} \mathit{Guard}(\text{ENTRY}_{c_i}, c_i)\Big)} \\ \mathit{IsFeasible}(\widetilde{\tau_u}) \end{array}}{\big((\widetilde{c_f}, \widetilde{\tau_f}), (\widetilde{c_u}, \widetilde{\tau_u})\big) \in \big(\widetilde{C}(l_f) \times \widetilde{\mathcal{G}}(l_f)\big) \times \big(\widetilde{C}(l_u) \times \widetilde{\mathcal{G}}(l_u)\big)}$$

Figure 9 illustrates the intraprocedural paths captured by these guards (marked by different colors). The interprocedural path from `xfree()` to the common caller and the interprocedural path from the common caller to `xuse()` are distinguished by calling contexts. ENTRY_{c_i} denotes the entry statement of the function containing the point c_i. Thus, $\widetilde{\tau_f}$ represents the path from the entry of the function containing the first call site in $\widetilde{c_f}$ to `free`$(p@l_f)$, and $\widetilde{\tau_u}$ for `use`$(q@l_u)$ consists of three parts: (i) $\widetilde{\tau_f}$, (ii) $\mathit{Guard}(l_f, \text{car}(\widetilde{c_u} \oplus [l_u]))$, which represents the path from l_f to the first call site in $\widetilde{c_u}$, and (iii) $\bigwedge_{c_i \in cdr(c_u \oplus [l_u])} \mathit{Guard}(\text{ENTRY}_{c_i}, c_i)$, which is similarly defined as $\widetilde{\tau_f}$. Given a sequence, *car* returns its first element and *cdr* returns the rest in the sequence. We also check the feasibility of $\widetilde{\tau_u}$ (and $\widetilde{\tau_f}$ implicitly) by using a SMT solver to enforce branch correlation.

$\mathbb{ST}^P$ is efficiently verifiable. For $\rightsquigarrow$, $(\widetilde{c_f}, l_f) \rightsquigarrow (\widetilde{c_u}, l_u) \implies ((\widetilde{c_f}, \widetilde{\tau_f}), l_f) \rightsquigarrow ((\widetilde{c_u}, \widetilde{\tau_u}), l_u)$. For $\cong$, we check $((\widetilde{c_f}, \widetilde{\tau_f}), p) \cong ((\widetilde{c_u}, \widetilde{\tau_u}), q)$ by querying $pt_2((\widetilde{c_f}, \widetilde{\tau_f}), p)$ and $pt_2((\widetilde{c_u}, \widetilde{\tau_u}), q)$.

Let us see how $\big($`free`$(\texttt{p@}ln4)$, `use`$(\texttt{p@}ln7)\big)$ in Figure 5 is reported as a UAF warning in Stage 1 (with calling-context reduction) but removed as a false alarm in Stage 2 (with path reduction). In Stage 1, $\widetilde{C}(ln4) \times \widetilde{C}(ln7) = \{([\,],[\,])\}$ by applying `[CTX-R]`. As $([\,], ln4) \rightsquigarrow ([\,], ln7)$ and $([\,], \texttt{p@}ln4) \cong ([\,], \texttt{p@}ln7)$ (since $pt_1([\,], \texttt{p@}ln4) = \{o_1\}$ and $pt_1([\,], \texttt{p@}ln7) = \{o_1, o_2\}$), a UAF warning is issued. Let us now apply `[PAT-R]`. We find that $\widetilde{\tau_f}$ = cnd encodes the path from the entry of the function `foo()` to line 4. Similarly, $\widetilde{\tau_u}$ = cnd∧true∧true = cnd encodes the path from the entry to line 7 via line 4. Thus, $(([\,],\text{cnd}), ln4) \rightsquigarrow (([\,],\text{cnd}), ln7)$. We obtain $\big(\widetilde{C}(ln4) \times \widetilde{\mathcal{G}}(ln4)\big) \times \big(\widetilde{C}(ln7) \times \widetilde{\mathcal{G}}(ln7)\big) = \big\{\big(([\,],\text{cnd}), ([\,],\text{cnd})\big)\big\}$. As $pt_2(([\,], \text{cnd}), p@ln4) = \{([\,], \text{cnd}, o_1)\}$ and $pt_2(([\,], \text{cnd}), p@ln7) = \{([\,],\text{cnd},o_2)\}$, we have $(([\,],\text{cnd}),\texttt{p@}ln4) \not\cong (([\,],\text{cnd}),\texttt{p@}ln7)$. Thus, $\big($`free`$(\texttt{p@}ln4)$, `use`$(\texttt{p@}ln7)\big)$ has been filtered out as a false alarm.

4 IMPLEMENTATION

We have implemented Stc in LLVM (3.8.0). The source files of a program are compiled under "-O0" into bit-code by `clang` front-end and then merged using the `LLVM Gold Plugin` at link time to produce a whole program bc file. For debugging purposes, LLVM under "-O1" or higher flags behaves non-deterministically on undefined (i.e., **undef**) values [45], making bug detection nondeterministic.

We have implemented our demand-driven pointer analysis, by operating on the def-use chains computed by the open-source software tool, SVF [36], field-sensitively but flow- and context-insensitively using Andersen's algorithm [4]. The wave propagation technique [28, 42] is used for constraint resolution. The positive weight cycles [27] are detected by Nuutila's algorithm [25]. Distinct allocation sites (i.e., AddrOf statements) are modeled by distinct abstract objects as in [12]. A program's call graph is built on the fly and points-to sets are represented using sparse bit vectors.

In static analysis, a linked list is modeled finitely. Thus, a node in a points-to cycle is not considered for UAF detection (to avoid false alarms), as it may represent many different concrete nodes.

Arrays must be approximated in static analysis. When computing $\cong$ with a pointer analysis, arrays are considered monolithic. When computing $\rightsquigarrow$, we distinguish different array elements intraprocedurally. LLVM's ScalarEvolution pass is applied to reason about must-aliases between two array accesses intraprocedurally.

Path guards are encoded by BBDs (Binary Decision Diagrams) using CUDD-2.5.0 [32]. For path feasibility, $\mathit{IsFeasible}(\widetilde{\tau_u})$ in `[PAT-R]` is checked by a SMT solver, known as Z3 [9].

5 EVALUATION

We show that Stc is efficient and effective in detecting UAF bugs in real-world programs without generating excessively many false alarms, by answering four research questions (RQs):

RQ1: Is Stc effective in detecting existing UAF bugs?
RQ2: Can Stc find (true) UAF bugs efficiently with a low false positive rate in programs with millions of lines of code?
RQ3: What are the patterns of false alarms eliminated?
RQ4: What are the patterns of UAF bugs detected?

5.1 Methodology

Stc is fully automatic without requiring user annotations. To answer RQ1 and RQ2, we compare Stc with four state-of-the-art automatic static tools: (1) CBMC (a bounded model checker for C/C++) [14], (2) Clang (an abstract interpreter for C/C++ in LLVM) [3], (3) Coccinelle (a pattern-based bug detector for C) [26], and (4) Supa (a flow- and context-sensitive demand-driven pointer analysis for C used for detecting UAF bugs according to $\mathbb{ST}^{\text{Supa}}$ in (2) [35]. To answer RQ3 and RQ4, we perform manual inspection in real code to check whether a reported UAF warning is a bug or not.

5.2 Benchmarks

Table 1: Benchmarks.

Program	Version	KLOC	#Pointers	#Frees	#Uses
bison	3.0.4	113	102679	299	20163
curl	7.52.2	188	16432	249	2179
ed	1.1	3	1062351	17	1604
grep	2.21	118	1692834	193	5910
ghostscript	9.14	1693	24067	489	255891
gzip	1.6	644	106458	66	3904
phptrace	0.3	6	354077	39	1344
redis	3.2.6	133	37793	782	59056
sed	4.2	38	548267	221	6969
zfs	0.7.0	327	52629	680	6162

To answer RQ1 (for ground truth), we use all the C test cases in Juliet Test Suite (JTS) [1], including 138 known UAF vulnerabilities. Each test case consists of 100 - 500 lines of code extracted from real-world applications. To answer RQ2 – RQ4 (in order to test the practicality of Stc), we use 10 widely-used open-source C applications, totaling over 3 MLOC, given in Table 1.

5.3 Experimental Setup

CBMC is configured to run as a UAF detector by enabling "–pointer-check" and disabling the other checks. To ensure that CBMC handles loops identically as Stc (as described in Section 3.1.2), every loop is unrolled by specifying "-unwind 2". To ensure termination, the per-program analysis budget for CBMC is set as 3 days. To use Clang, each program is compiled with "scan-build ./configure" and "scan-build make", following its official user manual [3]. Coccinelle is invoked with `spatch --sp-file`, with the UAF patterns specified with its official UAF script, `osdi_kfree.cocci`. Supa is used for finding UAF bugs according to the analysis given in (2).

Both Supa and Stc share the same pre-analysis, which is performed with Andersen's algorithm [4] field-sensitively but flow-, context- and path-insensitively. In both cases, the budget for one points-to query is set as 300,000 (the maximum number of def-use chains traversable). For any larger budget, both Supa and Stc take longer to run but exhibit small improvements in precision.

For Stc, we apply one optimization in Stage 2 to reduce the human effort required in inspecting warnings. Consider two warnings, $B_1 = \left(\texttt{free}(p@l_f), \texttt{use}(q_1@l_u^1)\right)$ and $B_2 = \left(\texttt{free}(p@l_f), \texttt{use}(q_2@l_u^2)\right)$, with the same free site. It suffices to report B_1 only if B_1 is a bug whenever B_2 is and B_2 is a false alarm whenever B_1 is. This happens if (1) $pt_2((\widetilde{c_u^1}, \widetilde{\tau_u^1}), q_1)$ includes all the objects in $pt_2((\widetilde{c_u^2}, \widetilde{\tau_u^2}), q_2)$ and (2) $\widetilde{\tau_u^2} \implies \widetilde{\tau_u^1}$ (solved by Z3).

Our experiments were done on a 3.0 GHZ Intel Core2 Duo processor with 128 GB memory, running RedHat Enterprise Linux 5 (2.6.18). The analysis time of a program is the average of 3 runs.

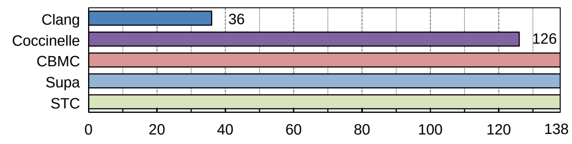


Figure 10: Hit rates for the 138 bugs in JTS: CBMC (100%), Clang (26%), Coccinelle (91%), Supa (100%), and Stc (100%).

5.4 Results and Analysis

5.4.1 RQ1: Recall (i.e., Hit Rate). We assess whether Stc is capable of locating the 138 known UAF bugs in JTS [1]. As displayed in Figure 10, Stc finds all the 138 bugs, just as CBMC and Supa do, but Clang and Coccinelle detect only 36 and 126 bugs, respectively, with no false alarms produced by any tool.

Stc achieves a total recall, i.e., a 100% hit rate in 3.7 seconds. CBMC, as a verification tool, also achieves a total recall but in 125.5 seconds, the longest among all the five tools. Supa, as a sound pointer analysis, achieves a total recall in 3.0 seconds.

Both Clang and Coccinelle miss some bugs. Clang finds only 36 bugs in 2.5 seconds with a hit rate of 26%. Clang fails to detect 102 out of 138 UAF bugs for several reasons: (i) it lacks a pointer analysis, (ii) it performs only some limited interprocedural analysis through inlining, and (iii) it reasons about loops very conservatively.

Coccinelle detects 126 bugs in 19.7 seconds. It has missed 12 bugs due to some unsound search-space reduction heuristics used. One is concerned with matching a free site with its use sites. Given a free site, Coccinelle examines only the use sites reachable along the forward edges in the program's call graph. Thus, any UAF bug will be missed if its free site resides in a wrapper. Another is related to the limited alias analysis in Coccinelle. Given a free site $\texttt{free}(p)$, Coccinelle considers the aliases for $*p$ by tracking only the value-flow of p forwards along the control flow via only a sequence of copy assignments on top-level variables. Thus, an alias between $*p$ and $*q$ (for a $\texttt{use}(q)$) that is formed before $\texttt{free}(p)$ or indirectly via address-taken variables in terms of loads and stores will be missed. All the 12 bugs in JTS are missed this way.

5.4.2 RQ2: Bug-Finding Ability. We assess how efficiently and effectively Stc finds new UAF bugs in the 10 real-world applications (Table 1). Table 2 gives the results. Stc issues 132 warnings including 85 bugs in 27,334 seconds (7.6 hours), starting from 41,843 warnings generated by the pre-analysis. However, the four existing tools are either unscalable by terminating within 3 days only for one application (CBMC) or impractical by reporting virtually no bugs (Clang and Coccinelle) or excessively many false alarms (Supa).

CBMC does not scale yet to large codebases. It spends 68,553 seconds, i.e., 19.0 hours in analyzing ed (the smallest with 3 KLOC) but cannot terminate for each remaining application in 3 days. As a result, CBMC detects no UAF bugs (as ed is absent of UAF bugs).

Clang reports 3 warnings including only 1 (intraprocedurally-detectable) bug, which is also found by Stc, in 1.2 hours. Interestingly, Clang exhibits even a higher false positive rate than Stc.

Coccinelle reports 103 warnings, which are all false alarms by manual inspection, in 179.0 seconds. Coccinelle fails to detect any true bug due to mainly its two unsound heuristics that are described above in Section 5.4.1. Specifically, among the 85 bugs detected by Stc, 84 bugs require tracking the backward (i.e., return)

Table 2: Experimental results (#T:#True Positives (Bugs) and #F: #False Positives (i.e., False Alarms)).

Program	Clang			CBMC			Coccinelle			Supa			Stc							
	Report		Time	Report		Time	Report		Time	Report		Time	#Warnings			Context Reduction		Report		Time
	#T	#F	(secs)	#T	#F	(secs)	#T	#F	(secs)	#T	#F	(secs)	Pre	Stage 1	Stage 2	Before	After	#T	#F	(secs)
bison	0	0	113	0	0	> 259200	0	18	7	0	1044	1793	1640	352	1	7.3×10^{15}	2.0×10^{5}	0	1	1904
curl	0	0	355	0	0	> 259200	0	8	53	0	694	27	699	82	0	3.3×10^{7}	8.2×10^{3}	0	0	668
ed	0	0	18	0	0	68553	0	0	1	0	34	1	34	32	2	6.3×10^{4}	3.6×10^{3}	0	2	4
grep	0	0	110	0	0	> 259200	0	18	9	1	537	362	630	493	2	1.1×10^{7}	3.0×10^{5}	1	1	2023
ghostscript	0	0	2007	0	0	> 259200	0	23	68	0	1944	2556	2630	1038	3	6.4×10^{15}	1.6×10^{5}	0	3	2805
gzip	1	0	68	0	0	> 259200	0	12	3	1	381	3	382	117	1	7.1×10^{8}	3.6×10^{3}	1	0	4
phptrace	0	0	29	0	0	> 259200	0	0	1	1	192	1	268	5	1	7.0×10^{5}	3.2×10^{3}	1	0	2
redis	0	2	836	0	0	> 259200	0	5	7	16	4187	13333	11019	395	20	1.1×10^{15}	4.0×10^{3}	16	4	13551
sed	0	0	116	0	0	> 259200	0	14	3	26	1887	160	2258	441	29	1.0×10^{9}	1.8×10^{5}	26	3	5102
zfs	0	0	790	0	0	> 259200	0	5	30	40	12195	180	22283	2730	73	2.3×10^{14}	1.0×10^{6}	40	33	1271
Total	1	2	4442	0	0	> 2401353	0	103	179	85	23095	18416	41843	5685	132	1.5×10^{16}	1.9×10^{6}	85	47	27334

edges of wrappers for free sites, with one exception in `gzip`, which, however, requires analyzing the aliasing relations for address-taken variables. In addition, all the 26 bugs in `sed` and 32 bugs in `zfs` also require the value flows of address-taken variables to be tracked.

Supa also starts with the same 41,843 UAF warnings pre-computed by Stc's pre-analysis. Being sound, Supa reports the same 85 bugs found by Stc but also 23,180 warnings (with both as expected). These false alarms are the spurious spatio-temporal correlations introduced in $\mathbb{ST}^{\text{Supa}}$ in (2), as motivated in Section 2.

Dynamic detectors have low false positive rates but may not find UAF bugs. We have run AddressSanitizer [30] on the 10 applications with "-O0 -fsanitize=address" enabled. Under the default test inputs provided, it reports 5 memory leaks in `ghostscript` and `grep` and 1 buffer overflow in `bison`, but no UAF bugs at all in about 6.5 mins.

Stc is effective in finding new UAF bugs in real-world applications. By examining manually the 132 warnings reported, we found 85 to be bugs and 47 to be false positives. These false alarms are issued due to mainly imprecise handling of complex path conditions (among others explained in Section 5.5). Clang finds only 1 bug in `gzip`, which is also found by Stc, among the 3 warnings reported. The other 2 warnings (in `sed`) are false alarms, due to its lack of pointer analysis. These 2 false alarms are not reported by Stc. Stc is also highly effective in filtering out false alarms in all its two stages. Let w_i be the warnings produced by Stage i. The false alarm elimination (FAE) rate at Stage i, where $1 \leqslant i \leqslant 2$, is given by $(w_{i-1} - w_i)/w_{i-1}$. The two Stc stages (Stages 1 and 2 in Table 2) are effective, with their average FAE rates being 68.7% and 95.8%.

Stc is also efficient in its two stages, as shown in Figure 11, by using increasingly more precise yet more expensive analyses on handling increasingly fewer UAF warnings (as validated in Table 2). In Stage 1 (context-sensitive analysis), context reduction is significant, as revealed in Columns 17 – 18 in Table 2. Otherwise, Stage 1 would run for 2×10^9 days for the 10 applications (estimated based on the per-query time consumed in each stage).

Given its effectiveness, Stc is the most scalable interprocedural UAF detector reported (to the best of our knowledge). Stc spends just 7.6 hours in analyzing the 10 applications (totaling 3+ MLOC). The analysis time for a program includes the times elapsed in its two stages and its pre-analysis. For Clang and Coccinelle, `ghostscript` takes the longest to analyze since it is the largest with 1693 KLOC. For Stc and Supa, `redis` takes the longest since it has the second largest number of UAF candidate pairs, i.e., 11,019 pairs to be analyzed and complex constraints to be solved by Z3.

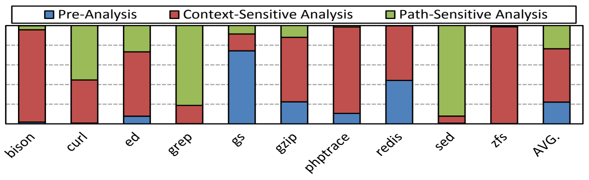


Figure 11: Percentage distribution of Stc's analysis times.

5.4.3 RQ3: False Alarm Patterns. To provide insights on detecting UAF bugs statically, we discuss some common patterns of false alarms removed by Stc in its two stages (Figures 12(a) and (b)).

Context Reduction (Stage 1). Figure 12(a) illustrates a false alarm in `gs` eliminated in Stage 1. Consider lines 250 and 3978. Supa (Table 2) will report $\big(\texttt{free(p@}\mathit{ln}250),\texttt{use(dir@}\mathit{ln}3978)\big)$ as a warning, since $([\,], \mathit{ln}250) \rightsquigarrow ([\,], \mathit{ln}3978)$ and $([\,], \texttt{p}) \cong ([\,], \texttt{dir})$ (due to $\mathit{pt}([\,], \texttt{p}) \cap \mathit{pt}([\,], \texttt{dir}) \neq \varnothing$). With calling-context reduction, this will be identified as a false alarm successfully.

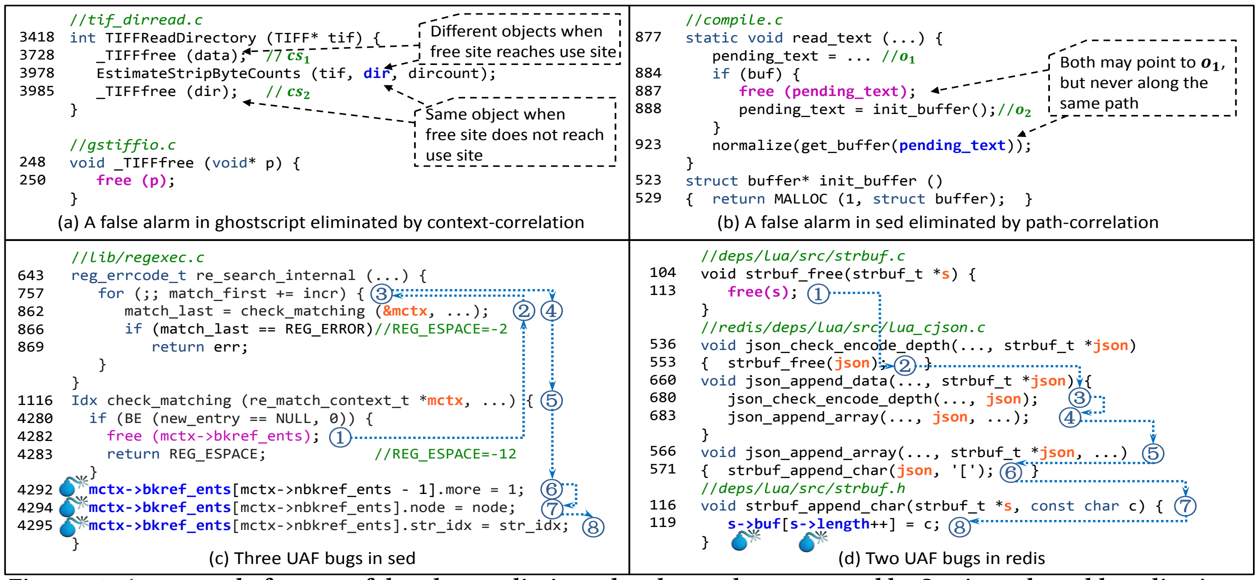
Path Reduction (Stage 2). Figure 12(b) gives a UAF candidate $\big(\texttt{free(pending\_text@}\mathit{ln}887),\texttt{use(pending\_text@}\mathit{ln}923)\big)$, that is identified as a false alarm path-sensitively (by capturing branch correlation), just like $\big(\texttt{free(p@}\mathit{ln}4),\texttt{use(p@}\mathit{ln}7)\big)$ in Figure 5.

5.4.4 RQ4: Understanding UAF Bugs. There are 85 UAF bugs detected by Stc. We first examine two representative patterns in Figures 12(c) and (d) and then discuss these bugs briefly.

Figure 12(c) illustrates three UAF bugs found in `sed` (counted as one in Column 19 in Table 2, as discussed in Section 5.3). Under a certain path condition, the program frees `mctx->bkref_ents` (line 4282) and returns an error flag `REG_ESPACE` (line 4283). Unfortunately, the error is not captured later in line 866, since `REG_ESPACE` $\neq$ `REG_ERROR`, causing the freed pointer `mctx->bkref_ents` to be dereferenced in lines 4292 – 4294.

Figure 12(d) gives two UAF bugs (counted as one in Column 19 in Table 2) in `redis`. In function `json_append_data` (line 660), `json` is indirectly freed in line 680 by calling `json_check_encode_depth`, which in turn calls `strbuf_free` (line 553) to free the object (line 113). After that, `json_append_data` calls `json_append_array` (line 683) with `json` passed as a parameter, where the freed object is accessed twice (line 119), resulting in two UAF bugs.

The 85 bugs detected by Stc reside in `grep`, `gzip`, `phptrace`, `redis`, `sed` and `zfs`. Pointer analysis is essential. As mentioned earlier, 58 bugs (including 26 in `sed` and 32 in `zaf`) require analyzing aliases for address-taken variables. The remaining 27 bugs, which are found in `grep`, `gzip`, `phptrace`, `redis` and `zfs`, require


Different objects when free site reaches use site
Same object when free site does not reach use site
Both may point to o_1, but never along the same path

```
//tif_dirread.c
int TIFFReadDirectory (TIFF* tif) {
   _TIFFfree (data);  //cs1
   EstimateStripByteCounts (tif, dir, dircount);
   _TIFFfree (dir);   //cs2
}

//gstiffio.c
void _TIFFfree (void* p) {
   free (p);
}
```

(a) A false alarm in ghostscript eliminated by context-correlation

```
//compile.c
static void read_text (...) {
   pending_text = ... //o1
   if (buf) {
      free (pending_text);
      pending_text = init_buffer();//o2
   }
   normalize(get_buffer(pending_text));
}
struct buffer* init_buffer ()
{  return MALLOC (1, struct buffer);  }
```

(b) A false alarm in sed eliminated by path-correlation

```
//lib/regexec.c
reg_errcode_t re_search_internal (...) {
   for (;; match_first += incr) {
      match_last = check_matching (&mctx, ...);
      if (match_last == REG_ERROR)//REG_ESPACE=-2
         return err;
   }
}
Idx check_matching (re_match_context_t *mctx, ...) {
  if (BE (new_entry == NULL, 0)) {
    free (mctx->bkref_ents);
    return REG_ESPACE;          //REG_ESPACE=-12
  }
mctx->bkref_ents[mctx->nbkref_ents - 1].more = 1;
mctx->bkref_ents[mctx->nbkref_ents].node = node;
mctx->bkref_ents[mctx->nbkref_ents].str_idx = str_idx;
}
```

(c) Three UAF bugs in sed

```
//deps/lua/src/strbuf.c
void strbuf_free(strbuf_t *s) {
   free(s);
}
//redis/deps/lua/src/lua_cjson.c
void json_check_encode_depth(..., strbuf_t *json)
{  strbuf_free(json); }
void json_append_data(..., strbuf_t *json) {
   json_check_encode_depth(..., json);
   json_append_array(..., json, ...);
}
void json_append_array(..., strbuf_t *json, ...)
{  strbuf_append_char(json, '['); }
//deps/lua/src/strbuf.h
void strbuf_append_char(strbuf_t *s, const char c) {
   s->buf[s->length++] = c;
}
```

(d) Two UAF bugs in redis

Figure 12: A case study for some false alarms eliminated and some bugs reported by Stc in real-world applications.

analyzing top-level pointers only. 4 bugs in zfs would be missied if some function pointers in the call sequence from their common callers to their use sites were not resolved accurately. In addition, interprocedural analysis is also essential. Consider Figure 8. The average call sequence from a common caller to a free (use) site is 2.33 (3.71), with the longest being 4 (7). For only one out of the 85 bugs, its free and use site reside directly in its common caller.

5.5 Limitations

As a static analysis, Stc can suffer from both false negatives and false positives. Stc can miss bugs due to its unsound modeling of loops (by analyzing two iterations), its unsound handling of a linked list (by ignoring its nodes participating in points-to cycles), and its unsound modeling of array access aliases (by using LLVM's ScalarEvolution pass for detecting must-aliases). In addition, in non-compliant C programs, where one uses a pointer pointing to one object to access another object with pointer arithmetic, pointer analysis will be unsound, resulting in potentially false negatives.

Stc yields false alarms due to mainly (i) imprecise path reduction in [PAT-R], and (ii) imprecise points-to information for out-of-budget points-to queries (in traversing points-to cycles).

6 RELATED WORK

In addition to Section 1, we now review the most relevant work on detecting and protecting against UAF bugs.

Detection. Almost all solutions are dynamic (instrumentation-based). Debugging tools such as Valgrind [23] and Dr.Memory [6] can detect a range of memory corruption errors including UAF bugs at the expense of high runtime and memory overheads. AddressSanitizer [30] is another widely used dynamic tool. However, it can miss dangling pointers that, when dereferenced, point to an object that has reused the memory range. Undangle [7] detects dangling pointers by performing a dynamic taint analysis. Its early detection approach can incur high runtime overheads. CETS [22] uses an identifier-based scheme, which assigns a unique key for each allocation region to identify dangling pointers. It has an overhead of 116% in order to provide complete memory safety.

We are unaware of static tools dedicated to UAF detection, for the reasons given in Section 1. General-purpose memory-safety checking tools that can be used to detect UAF bugs include CBMC [14], Clang [3], Coccinelle [26], and Supa [35], which have been compared with Stc. Specialized tools for detecting other types of bugs exist. Saturn [10, 40] detects memory leaks and null pointers by solving a Boolean satisfiability problem. FastCheck [8] and Saber [37] find memory leaks on the value-flow graph of a program. Buffer overflows can be detected path-sensitively [16] or symbolically [19].

Protection. Instead of detecting UAF bugs, protection against their exploitation can be made. For example, control flow integrity [11] prevents control-flow hijacking attacks due to UAF buffer overflow exploits via runtime instrumentation. However, all fine-grained solutions are too costly to be deployed in production environments and all coarse-grained solutions are bypassable [11].

Cling [2] represents a safe memory allocator that restricts memory reuse to objects of the same type. Diehard [5] and Dieharder [24] apply a randomized memory allocator by providing probabilistic safe guarantees. In these cases, UAF exploits are made harder but not eliminated. Alternatively, FreeSentry [43] and DangNull [17] invalidate the dangling pointers detected at runtime, at the expense of high runtime and memory overheads.

7 CONCLUSION

We present Stc, a novel static detector for finding UAF bugs, and demonstrate its effectiveness and efficiency in finding all the known UAF bugs in Juliet Test Suite and new ones in multi-MLOC C applications. To the best of our knowledge, Stc is the first such interprocedural analysis achieving this level of scalability, precision and accuracy. Its success relies on the three advances made: (i) a context reduction technique for scaling Stc to large codebases, (ii) a multi-stage approach for filtering false alarms earlier, and (iii) a field-, flow-, context- and path-sensitive demand-driven pointer analysis for providing the precise points-to information required.

In future work, we plan to address some of Stc's limitations stated in Section 5.5. We also plan to combine Stc with dynamic analysis to further filter out false alarms reported by Stc.

REFERENCES

[1] Juliet Test Suite 1.2. https://samate.nist.gov/SRD/testsuite.php.

[2] Periklis Akritidis. 2010. Cling: a memory allocator to mitigate dangling pointers. In *USENIX Security '10*. 177–192.

[3] Clang Static Analyzer. http://clang-analyzer.llvm.org/.

[4] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. DIKU, University of Copenhagen.

[5] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: probabilistic memory safety for unsafe languages. In *PLDI '06*. 158–168.

[6] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *CGO '11*. 213–223.

[7] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA ' 12*. 133–143.

[8] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07*. 480–491.

[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS '08*. 337–340.

[10] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *PLDI '08*. 270–280.

[11] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: on the weaknesses of fine-grained control flow integrity. In *CCS '15*. 901–913.

[12] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *CGO '11*. 289–298.

[13] Nevin Heintze and Olivier Tardieu. 2001. Demand-Driven Pointer Analysis. In *PLDI '01*. 24–34.

[14] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C Bounded model checker. In *TACAS'14*. 389–391.

[15] William Landi and Barbara G Ryder. 1992. A safe approximate algorithm for interprocedural aliasing. In *PLDI '92*. 235–248.

[16] Wei Le and Mary Lou Soffa. 2008. Marple: a demand-driven path-sensitive buffer overflow detector. In *FSE '08*. 272–282.

[17] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing use-after-free with dangling pointers nullification.. In *NDSS '15*.

[18] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *POPL '11*. 3–16.

[19] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2010. Practical and effective symbolic analysis for buffer overflow detection. In *FSE '10*. 317–326.

[20] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE '11*. 343–353.

[21] Ravichandhran Madhavan and Raghavan Komondoor. 2011. Null dereference verification via over-approximated weakest pre-conditions analysis. In *OOSPLA '11*. 1033–1052.

[22] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *ISMM '10*. 31–40.

[23] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*. 89–100.

[24] Gene Novark and Emery D Berger. 2010. DieHarder: securing the heap. In *CCS '10*. 573–584.

[25] Esko Nuutila and Eljas Soisalon-Soininen. 1994. On finding the strongly connected components in a directed graph. *Inform. Process. Lett.* 49, 1 (1994), 9–14.

[26] Mads Chr Olesen, René Rydhof Hansen, Julia L Lawall, and Nicolas Palix. 2014. Coccinelle: Tool support for automated CERT C secure coding standard certification. *Science of Computer Programming* 91 (2014), 141–160.

[27] David J Pearce, Paul HJ Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems* 30, 1 (2007), 4.

[28] Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave propagation and deep propagation for pointer analysis. In *CGO '09*. 126–135.

[29] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*. 49–61.

[30] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *USENIX ATC'12*. 309–318.

[31] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *CGO '12*. 264–274.

[32] Fabio Somenzi. CUDD: CU Decision Diagram Package (3.0.0). http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf.

[33] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: demand-driven flow-and context-sensitive pointer analysis for Java. In *ECOOP'16*. 22:1–22:26.

[34] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *PLDI '16*. 387–400.

[35] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *FSE '16*. 460–473.

[36] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. https://github.com/unsw-corg/SVF. In *CC '16*. 265–266.

[37] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*. 254–264.

[38] Kostyantyn Vorobyov and Padmanabhan Krishnan. 2010. Comparing model checking and static program analysis: a case study in error detection approaches. In *SSV*. 1–7.

[39] National vulnerability database. http://nvd.nist.gov/.

[40] Yichen Xie and Alex Aiken. 2007. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems* 29, 3 (2007), 16.

[41] Wei Xu, Daniel C DuVarney, and R Sekar. 2004. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *FSE '12*. 117–126.

[42] Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-based selective flow-sensitive pointer analysis. In *SAS '14*. 319–336.

[43] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS '15*.

[44] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*. 218–229.

[45] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL '12*. 427–440.

[46] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *POPL '08*. 197–208.