Canary: Practical Static Detection of Inter-thread Value-Flow Bugs

Yuandao Cai
The Hong Kong University of Science and Technology
Hong Kong, China
ycaibb@cse.ust.hk

Peisen Yao
The Hong Kong University of Science and Technology
Hong Kong, China
pyao@cse.ust.hk

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

Abstract

Concurrent programs are still prone to bugs arising from the subtle interleavings of threads. Traditional static analysis for concurrent programs, such as data-flow analysis and symbolic execution, has to explicitly explore redundant control states, leading to prohibitive computational complexity.

This paper presents a value flow analysis framework for concurrent programs called Canary that is practical to statically find diversified inter-thread value-flow bugs. Our work is the first to convert the concurrency bug detection to a source-sink reachability problem, effectively reducing redundant thread interleavings. Specifically, we propose a scalable thread-modular algorithm to capture data and interference dependence in a value-flow graph. The relevant edges of value flows are annotated with execution constraints as guards to describe the conditions of value flows. Canary then traverses the graph to detect concurrency defects via tracking the source-sink properties and solving the aggregated guards of value flows with an SMT solver to decide the realizability of interleaving executions. Experiments show that Canary is precise, scalable and practical, detecting over eighteen previously unknown concurrency bugs in large, widely-used software systems with low false positives.

CCS Concepts: • **Software and its engineering → Software verification and validation**.

Keywords: Concurrency, static analysis, interference dependence, concurrency bugs detection

ACM Reference Format:
Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: Practical Static Detection of Inter-thread Value-Flow Bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454099

1 Introduction

Hunting concurrency bugs is notoriously difficult with the explosive growth of code size and complexity in modern software. Dynamically detecting such errors is hard because it depends on intricate sequences of low-probability concurrent events [7, 19, 21, 28, 56]. The number of thread interleavings and feasible program paths grows exponentially with the number of threads and the length of program execution, making dynamic analysis difficult to exercise even a tiny fraction of all possible execution, leaving large systems with an abundance of errors [64]. Comparatively, static analysis achieves good coverage, discovering errors in obscure code paths difficult for the runtime execution to reach. Unfortunately, static analysis suffers considerably from being imprecise as a result of abstracting and approximating the runtime behavior of the program.

The precise static analysis of concurrent programs is particularly challenging, as it needs to account for the thread interleavings together with the complicated sequential reasoning. Techniques of either the conventional data-flow anlaysis [12, 17, 18, 35, 54] or symbolic execution [4, 23, 57] propagate the data-flow facts along the control-flow paths, which, unfortunately, are often redundant and irrelevant to the data-flow facts of interest when interleaving comes into play. With this inherent limitation of the state space explosion, these techniques often severely suffer from performance issues when aiming for high precision. In practice, practitioners have little tolerance to this limitation, because compromising either precision or scalability forms significant hurdles of adoption.

To reduce the exponential space, one promising direction is to identify the interference dependence [27, 46, 47], which is the additional data dependence in concurrent programs that takes place at shared memory locations between threads. Capturing the def-use relations between statements across different threads enables the effective pruning of the redundant interleaving. However, identifying the interference dependence itself is extremely challenging as witnessed

PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8391-2/21/06...$15.00
https://doi.org/10.1145/3453483.3454099

by a long stream of effort [46, 47, 51, 52, 54, 60]. The key difficulty is the precise pointer aliasing in the presence of thread interference, because exhaustively and precisely computing pointer information has not yet been proved scalable [54, 59–61]. On the other hand, it is also unnecessary to explore the interleaving space of a given program irrelevant to the specific properties being checked.

To address these problems, we present Canary, a novel concurrency analysis technique that tracks how values are stored and loaded via both the data and interference dependence relationships, thus able to check a diversified category of multi-threaded software safety properties. Our key insight to mitigate the state space explosion is to reduce concurrency bug detection to the on-demand tracking of the source-sink value flows. We extend the conventional notion of source-sink properties to include values that flow across threads through loads and stores and refer to the bugs detected this way as the *inter-thread value-flow bugs*, such as the inter-thread use-after-free [2, 7, 28], NULL pointer dereference [19], double-free [7] and information leak [21]. Our checking of value flows only follows the def-use relationships between the relevant statements across threads and, hence, avoids explicitly enumerating all the possible thread interleavings.

As illustrated in Fig. 1, Canary has three key phases. It first performs a thread-modular algorithm to separately capture both the data dependence and the interference dependence in a value-flow graph (VFG). Unlike the previous interference dependence analysis [60, 61], the proposed algorithm captures the interference dependence without performing the exhaustive pointer analysis as a prior. Instead, it piggybacks the pointer analysis with the resolution of the interference dependence. To represent the condition under which the value flows qualify for a feasible execution, the analyses encode and annotate all necessary execution constraints [29, 31] as guards on relevant edges of the VFG, following the definition of sequential consistency [36]. The edges of the interference dependence act as "tunnels" to allow the values of interests to flow in and out of the thread scope during the on-demand value flow searching. At the bug checking stage, Canary checks the source-sink properties using the interference-aware VFG by extracting the source-sink paths and proceeds to verify their realizability (i.e., whether violate a feasible execution) by feeding the collected constraints on the aggregated guards to a dedicated SMT solver.

The noteworthy features of Canary are as follows. First, it focuses on both the graph exploration and the SMT solving effort around the source-sink value flows that only matter to the property to be checked. Second, it integrates and solves the guard information that is just sufficient to characterize an interleaving execution, judiciously delaying the disjunctive reasoning of the realizability of the vulnerable paths until the SMT solving stage. Third, the use of value flows generates concise bug reports with a limited number of relevant statements and conditions that cause the concurrency errors, greatly helping to diagnose the root causes of the bugs.

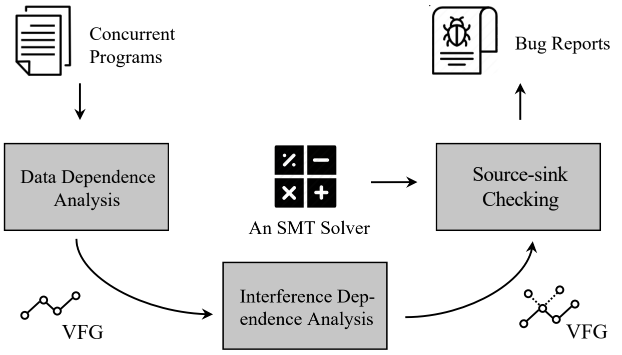


Figure 1. The workflow of Canary.

We have used Canary to check critical concurrency memory-safety properties, such as the inter-thread use-after-free and NULL pointer dereference, on a large set of popular open-source multi-threaded C/C++ software systems. Interestingly, although these software systems have been well examined by a crowd of both free and commercial tools, we are still able to discover over eighteen previously-unknown concurrency bugs confirmed by developers with fourteen of them fixed. Our evaluation shows that Canary has good scalability as it can build a highly precise value-flow graph up to >70× and >500× faster with less memory space, compared to the state of the arts, namely Saber [61] and Fsam [60]. Moreover, it is able to complete the path-sensitive checking of MySQL (around 3 MLoC code) within approximately 2.5 hours, the best in terms of scalability and precision, to the best of our knowledge.

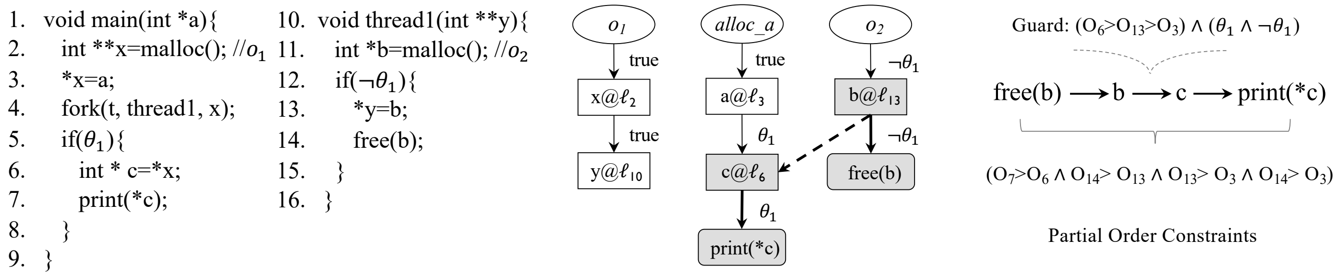
To sum up, the contributions of this work are as follows.

- An interference-aware value-flow analysis technique for modelling the checking diversified concurrency bugs as source-sink reachability problems.
- A thread-modular algorithm to identifying data dependence and interference dependence for concurrent programs.
- An implementation and an experiment that demonstrate Canary's good scalability as well as precision relative to previous techniques.

2 Canary in a Nutshell

We use a bug-free program in Fig. 2 to illustrate the concept of the thread interference on the shared memory locations, to clarify the shortcomings of the previous work, and then to highlight the essence of our approach.

The code snippet in Fig. 2(a) is bug-free, but *an inter-thread use-after-free bug* can be erroneously reported by previous techniques. In this program, the thread *main* creates



```
1.  void main(int *a){            10. void thread1(int **y){
2.    int **x=malloc(); //o1      11.   int *b=malloc(); //o2
3.    *x=a;                       12.   if(¬θ1){
4.    fork(t, thread1, x);        13.     *y=b;
5.    if(θ1){                     14.     free(b);
6.      int * c=*x;               15.   }
7.      print(*c);                16. }
8.    }
9.  }
```

o_1
true
x@ℓ_2
true
y@ℓ_{10}
$alloc\_a$
true
a@ℓ_3
θ_1
c@ℓ_6
θ_1
print(*c)
o_2
$\neg\theta_1$
b@ℓ_{13}
$\neg\theta_1$
free(b)
Guard: (O_6>O_{13}>O_3) ∧ (θ_1 ∧ $\neg\theta_1$)
free(b) ⟶ b ⟶ c ⟶ print(*c)
(O_7>O_6 ∧ O_{14}> O_{13} ∧ O_{13}> O_3 ∧ O_{14}> O_3)
Partial Order Constraints

(a) Code snippet (b) A value-flow graph (c) A source-sink path

Figure 2. Figure (a) is a bug-free code snippet. Figure (b) is its value-flow graph with some nodes and conditions omitted. (*alloc_a* denoted the memory object pointed to by *a*) Figure (c) is an extracted source-sink path.

a thread t that runs the function *thread1*. If ignoring the path conditions, the "bug" is triggered due to the thread interference: when the "freed" pointer b is stored to the memory object o_1 via a pointer y at Line 13 in the function *thread1*, propagated to the load statement at Line 6, and dereferenced eventually at Line 7. However, since the two path conditions at Line 13 and Line 6 contradict each other, this "bug" never surfaces to bite. Previous techniques for concurrent programs that do not understand path conditions can report this as a false positive.

Unfortunately, the exponential number of states induced by thread interleavings pose unique challenges to remaining path-sensitivity. In the value-flow analysis framework of Canary, the state space of all possible interleavings is reduced via capturing the interference dependence, formally, defined below.

Definition 1. *A statement s_1 is interference dependent on a statement s_2 if a definition of x at s_1 is used at s_2, and s_1 and s_2 concurrently execute.*

To avoid the false positive reported in Fig. 2(a), Canary first performs a thread-modular algorithm to construct the value-flow graph for the program shown in Fig. 2(b), in which, the algorithm begins with an intra-thread analysis to resolve the data dependence relations, followed by an inter-thread interference dependence analysis. The main idea to identify the interference dependence is to discover the pointers that point to the shared memory locations. Let ℓ_1 represent the statement at Line 1. For instance, after resolving the intra-thread data dependence relations (solid lines), the VFG encodes that variables x and y may point to the shared memory object o_1, since both are reachable from o_1. This enables the discovery of the interference dependence edge (dashed line), because $b@\ell_{13}$ may be stored to o_1 and $c@\ell_6$ may be loaded from o_1 afterwards. Next, we encode the execution constraints on the guard of the edge that represent the sequential consistency axioms for validating the realizability of value flows (explained in § 3). The details of the algorithm are explained in § 4.

After constructing the VFG, to find this "bug", the analyses only need to traverse the graph starting from the node *free*(b) as a source to find the use site *print*($*c$), a sink, along the def-use relations. Fig. 2(c) shows the extracted source-sink path and its simplified constraints.

Take the edge from $b@\ell_{13}$ to $c@\ell_6$ as an example. The guard of this edge is the conjunction of (1) the conditions under which the value flows through the same memory object o_1, and (2) the conditions enforcing that the value loaded from a load statement (ℓ_6) is the value stored by the most recent store statement (ℓ_{13}). The first conditions $\theta_1 \land \neg\theta_1$ include conjuncting the conditions for x and y pointing to an object o_1 ($true$), and the branch conditions of statements ℓ_6 and ℓ_{13} ($\theta_1 \land \neg\theta_1$). Let $O_2 > O_1$ be the *strict* partial order between statements ℓ_1 and ℓ_2 that indicates "ℓ_2 is executed after ℓ_1". The second conditions $O_{13} < O_6$ indicate ℓ_{13} should be executed before the ℓ_6. We also include load-store order $O_3 < O_{13}$ to ensure that another store statement at ℓ_3 can not happen between lines ℓ_{13} and ℓ_6, making it possible to flow from $b@\ell_{13}$ to $c@\ell_6$ without being overwritten by $a@\ell_3$.

Because the interference dependence relations are not transitive [46, 47], this value-flow path may violate the program order during searching. Our analysis lazily encodes the partial order constraints over the path at the bug checking stage, taking the correct program order into account. For example, we have intra-thread order $O_7 > O_6 \land O_{14} > O_{13}$, following the order of control flows. We also have inter-thread order $O_{13} > O_3 \land O_{14} > O_3$, following fork/join synchronization semantics, since the thread t is forked at ℓ_3 and all statements in the thread t should happen after ℓ_3. Finally, all constraints are handed to an SMT solver, and Canary does not report the bug when the aggregated execution constraints are unsatisfiable. Note that a source-sink path with relevant statements is very useful for debugging a concurrency error. Also, the analyses skip the constraints irrelevant to the source-sink properties, saving a significant computational overhead.

The example above illustrates how Canary works in a nutshell. Seemingly redundant, the order constraints, as a part of execution constraints, are important to achieve precision by validating whether a value-flow path is realizable at the runtime. In § 3.2, we give details of all necessary execution constraints, following the sequential consistency axioms.

3 Preliminaries

In this section, we present the basic terminologies and notations, and formalize the key problem when considering interference dependence.

3.1 Bounded Concurrent Programs

We first make a standard assumption about the analyzed concurrent programs. We assume that the shared memory system is sequentially consistent [36], where memory operations are executed in the order in which they appear in the program, and every memory operation is atomic. Sequential consistency is one of the strongest memory models discussed in the literature. This assumption matches the natural expectation of programmers that a program behaves as an interleaving of the memory accesses from its constituent threads.

Language. We formalize our approach using a simple call-by-value language in Fig. 3, similar to the previous work [38, 39, 55]. Note that it is known to be undecidable to analyze arbitrary concurrent programs over infinite threads and data types. Our analyses gain decidability by structurally bounding the concurrent programs by unrolling both loops and recursive functions to a finite depth, which not only indirectly fixes the number of threads but also fixes the size of the heap that the bounded program may access. None of these aspects is handled by most previous static concurrent analysis techniques [3, 12, 57, 60, 64].

Abstract Domains. The symbols and abstract domains are listed in Fig. 4. A thread id $t \in \mathcal{T}$ represents a thread that corresponds to a context-sensitive fork site and comprises a finite number of functions F^+. A label $\ell \in \mathcal{L}$ indicates the program position of a statement in the control flow graph (CFG). We follow the LLVM convention of separating variables into two disjoint sets of top-level $v \in \mathcal{V}$ and address-taken variables $o \in \mathcal{O}$ as in the previous work [24, 37–39, 60]. A top-level variable v can point to different memory objects o on different guard conditions φ. The address-taken variables $\mathcal{O}$ are indirectly accessed via load and store statements, which take the top-level pointer variables, $\mathcal{V}$, as arguments.

Note that, the four-types of pointer operations in Fig. 3 are in the partial SSA form, and only the address-taken variables, $\mathcal{O}$, can be shared among threads and accessed via load or store statements. The nested pointer dereferences are eliminated by introducing the auxiliary variables [37], ensuring each load and store statement is counted as at most one shared access.

$$
\begin{array}{rl}
\text{Program } P & := F^+ \\
\text{Function } F & := func(v_1,\ldots,v_n)\{S^*;\} \\
\text{Statement } S & := \\
 & \mid v_1 = v_2 \mid v_1 = \&v_2 \\
 & \mid v_1 = *v_2 \mid \; * v_1 = v_2 \\
 & \mid v_1 = v_2 \text{ binop } v_3 \mid v_1 = \text{unop } v_2 \\
 & \mid if\,(v)\ then\ S_1\ else\ S_2 \mid return\ (x_0,\ldots,x_n) \\
 & \mid (x_0,\ldots,x_n) = call\ f(v_1,\ldots,v_n) \mid S_1; S_2 \\
 & \mid fork(t,f) \mid join(t) \\
\text{binop} & := + \mid - \mid \wedge \mid \vee \mid > \mid = \mid \neq \mid \cdots \\
\text{unop} & := - \mid \neg \mid \cdots
\end{array}
$$

Figure 3. The syntax of the language.

$$
\begin{array}{ll}
\text{Threads } t \in \mathcal{T} & \text{Labels } \ell \in \mathcal{L} \\
\text{Objects } o \in \mathcal{O} & \text{Variables } v \in \mathcal{V}
\end{array}
$$

Figure 4. The abstract domains.

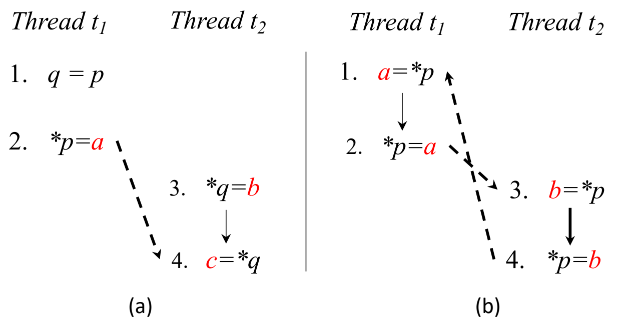


Figure 5. Two examples of concrete interleaving executions.

Value Flows. We say the value of a variable, a, flows to a variable, b, if a is assigned to b directly (via assignments, such as $b = a$) or indirectly (via pointer dereferences, such as $*y = a; x = y; b = *x$). The relations between a and b are either data dependence or interference dependence, depending on if the value flow happens across threads. A value-flow graph can be defined as a directed graph. Each node n is indicated by $v@\ell$, meaning that the variable v is defined or used at the program location ℓ. An edge that indicates the value-flow relation can be represented as a tuple, $(v_1@\ell_1, v_2@\ell_2)$. The guard Φ_{guard} annotated on the edge indicates the condition under which the value flow happens. A path $\pi = \langle v_1@\ell_1, \ldots, v_n@\ell_n \rangle$ on the value-flow graph is called the value-flow path.

Direct flows can be easy to identify based on the partial SSA form of statements. Therefore, we focus on indirect flows from the store statements to load statements via either the data dependence or the interference dependence, which requires the reasoning about the pointer information.

3.2 The Realizable Value-flow Path Problem

When considering the interference dependence, to obtain a precise value-flow path, it is necessary to ensure that the path can correspond to a feasible interleaving execution. We call this *the realizable value-flow path problem*. Ideally, one would like to compute a value-flow path that is the most precise. But this is, generally, not computable [50]. We give a practical and precise solution to characterize this problem, which guides our subsequent approaches.

Formally, we use the definition of *sequential consistency* as the axiomatic foundations to characterize a value-flow path. Let $<_P$ be the program order.

Definition 2. *A value-flow path $\pi = \langle v_1@\ell_1, \dots, v_n@\ell_n\rangle$ is said to be sequentially consistent if there exists a total order $<_\pi$ over the program order $<_P$ subject to the following conditions:*

1. *Each $v_{i-1}@\ell_{i-1}$ flows to $v_i@\ell_i$ through the same memory location without being overwritten, where $i \in [2, n]$.*
2. *For any ℓ_i and ℓ_j, if $\ell_i <_P \ell_j$, then $\ell_i <_\pi \ell_j$, i.e., $<_P \Rightarrow <_\pi$.*

First of all, the imprecise dependence relations can lead to an irrealizable value-flow path π, i.e., the stored variable $v_{i-1}@\ell_{i-1}$ may not flow to the loaded variable $v_i@\ell_i$ indirectly as outlined in Defn. 2(1). Compared to the conventional value-flow techniques for sequential programs, considering the interference dependence introduces additional challenges. The analysis must cut through the tangle of the additional aliasing information to reason about interference dependence associated with the non-deterministic concurrent environment. For example, in Fig. 5(a), the solid (dashed) lines represent possible data (interference) dependence, and the order of statements represents a concrete execution order. A variable $a@\ell_2$ can flow to $c@\ell_4$ only with the thread schedule, in which the statement ℓ_4 happens after ℓ_2 and the statement ℓ_3 happens before ℓ_2. As a result, the analysis should take the order relations between the loads and the stores into account in reasoning about alias.

Second, even if a value-flow path π is identified with precise dependence relations, it may not be realizable if violating $<_P \Rightarrow <_\pi$ according to Defn. 2(2). The $<_\pi$ represents the total order between each pair of successive nodes in π, while $<_P$ represents the program order, including both the control flow and the synchronization semantics. In Fig. 5(b), a value-flow path $\langle a@\ell_2, b@\ell_3, b@\ell_4, a@\ell_1\rangle$ is not realizable, because the value-flow path leads to an invalid control-flow path from ℓ_2 to ℓ_1 where $<_P \not\Rightarrow <_\pi$. The innate reason is the intransitive property of interference dependence [46, 47]: if a node n_i is interference dependent on a node n_j, and n_j is interference dependent on node n_k, it does not directly imply an interference dependence from n_i to n_k. Therefore, the arbitrary compositions of interference dependence relationships result in irrealizable value-flow paths. As a result, the analysis should take the program order into account while searching for value-flow paths.

In § 4, we present an algorithm to construct the value-flow graph with the resolved indirect flows from the store statements to the load statements. For the value flow from $q@\ell_1$ to $p@\ell_2$ between statements $\ell_1 : *x = q$ and $\ell_2 : p = *y$, the key is to decide if x is an alias of y. The main idea is to capture the escaped objects first and then discover all pointer variables that can point to these objects. The analysis encodes the necessary conditions of the sequentially consistency axioms as guards Φ_{guard}, including (1) (Memory Alias) the condition under which the variables x and y may point to the same memory object, and (2) (Load-store Order) the condition under which $q@\ell_1$ can flow to $p@\ell_2$ through the same memory object without being overwritten by other concurrent store statements, following Defn. 2(1). At the bug checking stage described in § 5, the partial order constraints Φ_{po} over the path are generated lazily, taking the program order $<_P$ into account that follows Defn. 2(2). In the end, the aggregated constraints enforcing the sequential consistency axioms, are handed to an SMT solver to validate if the value-flow path enforces a feasible execution.

4 Thread-modular Dependence Analysis

This section covers the details of the thread-modular algorithm, which resolves the data dependence and the interference dependence relations in the VFG by the alias analysis. Our algorithm consists of both the intra-thread and the inter-thread data dependence analysis phases, computing the conditions of dependence on edges as the guards.

4.1 Data Dependence Analysis

At the stage of the intra-thread reasoning, the goal is to resolve the intra-thread data dependence edges and compute their guards Φ_{guard}, following Defn. 2(1). Fig. 6 shows the basic rules for resolving the data dependence. The core problem is to calculate indirect flows from $q@*x = q$ to $p@p = *y$, which needs the points-to facts of the address-taken variables O. The Φ_{guard} of the data dependence at this stage is the conjunction of the conditions under which x and y point to the same memory object o, together with the branch conditions of the statements ℓ_1 and ℓ_2. The VFG generated at this stage serves as an input to bootstrap the interference dependence analysis.

More specifically, to decompose the expensive cost of the exhaustive points-to analysis, we follow the similar ideas as our previous work [55] by performing the intra-procedural path-sensitive points-to analysis to resolve the local data dependence relationships. The points-to analysis is performed along with a thread call graph in a bottom-up fashion, during which the analyses calculate the pointer information of the address-taken variables, O, and resolve the local data dependences in the VFG. A thread call graph is an extension of the call graph for sequential programs by collecting the call graphs of a group of threads at their fork sites. We next give

Instruction Code	Value Flow	Guard Φ_{gaurd}
$\ell, \varphi : p = alloc\_o$	$alloc\_o \rightarrow p$	φ
$\ell, \varphi : p = q$	$q \rightarrow p$	φ
$\ell_1, \varphi_1 : *x = q;$ $\ell_2, \varphi_2 : p = *y$	$\exists\, o \mid (o, \alpha) \in Pts(x) \wedge (o, \beta)$ $\in Pts(y) : q \rightarrow p$	$\varphi_1 \wedge \varphi_2 \wedge \alpha \wedge \beta$

Figure 6. Basic rules for data dependence.

the details of the intra-procedural points-to computation and the method to handle side-effects incurred by function calls.

Intra-procedural Analysis. The intra-procedural analysis is a fairly standard data-flow analysis [24] that explores the intra-procedural control flow graph in the reverse post-order while computing the points-to facts, *Pts*, at each statement and propagating these facts to the successors. We first perform a standard transformation (Line 3) to the function F by introducing auxiliary variables for the objects passed into the function by references, which is applied to explicitly expose the side-effects on the function's parameters [38, 39, 55]. Because the variables $\mathcal{V}$ are in SSA form, the analyses directly use a global points-to graph PG_{top} to hold the pointer information for the top-level variables, $\mathcal{V}$, and use the sets IN_ℓ and OUT_ℓ to hold the propagated pointer information for the address-taken variables, $\mathcal{O}$, at the statement ℓ [24]. During the calculation of the four-typed pointer operations (Line 11-20), the points-to conditions are consecutively recorded without being solved aggressively. After disambiguating the points-to information of the address-taken variables, $\mathcal{O}$, the indirect flows can be discovered based on the rules in Fig. 6.

Procedural Transfer Function. The procedural transfer function for a function, $Trans(F)$, summarizes its points-to side-effects (i.e., changes) between its formal-in parameters and formal-out parameters (Lines 21-22), which expresses the function's behavior in terms of its input/output interfaces, abstracting away from its internal details. When analyzing a call site, the analysis reasons about the callee's side-effects on actual parameters by applying its procedural transfer function, which bears some similarities with the work [16, 68]. However, there is no need for an equivalent reasoning for the parameters passed to the fork sites (Lines 23-24), since the interference dependence incurred by the parameters is going to be identified afterwards.

Example 4.1. In Fig. 2(b), the solid lines represent the data dependence identified at this stage. The VFG at this stage contains the important pointer information for the subsequent interference dependence analysis. For instance, we can identify the escaped objects, i.e., the shared memory locations, by finding the memory objects in the VFG that are reachable from nodes in different threads, and collect the pointer variables pointing to them. In the example, the analyses can find an escaped object o_1 and two pointer variables x and y that point to o_1 by traversing the VFG.

Algorithm 1: Data Dependence Analysis

```
Input: A concurrent program $CP$
Output: $VFG$ with resolved intra-thread data dependence
1  $CG \hookleftarrow$ construct the thread call graph;
2  foreach function $F$ in reverse topological order of $CG$ do
3      Transform $F : func(v_1, \ldots, v_m) \Rightarrow (v_1, \ldots, v_n)S^*$;
4      $CFG \hookleftarrow$ Build intra-procedural control flow graph of $F$;
5      foreach instruction $I$ in reverse post-order of $CFG$ do
6          HandleEachInst($I$);
7      Record side-effects on $v_i$ and construct $Trans$ for $F$;
8      Resolve local data dependence relations in $VFG$;
9  return $VFG$;
10 Procedure HandleEachInst($I$):
11     if $\ell, \varphi : p = alloc\_o$ : then
12         $PG_{top} \hookleftarrow \{p \rightarrowtail (\varphi, alloc\_o)\}$ ; // $\rightarrowtail$: point to
13     if $\ell, \varphi : p = q$ : then
14         $PG_{top} \hookleftarrow \{p \rightarrowtail (\gamma \wedge \varphi, o)\}, \forall(\gamma, o) \in Pts(q)$;
15     if $\ell, \varphi : *x = q$ then
16         if $Pts(x)$ is a singleton then
17             $IN_\ell \hookleftarrow (IN_\ell \backslash Pts(x))$;
18         $OUT_\ell \hookleftarrow IN_\ell \cup \{Pts(x) \rightarrowtail (\gamma \wedge \varphi, o)\}, \forall(\gamma, o) \in Pts(q)$;
             Propagate $OUT_\ell$ to the successor(s);
19     if $\ell, \varphi : p = *y$ then
20         Let $\mathcal{P}_{adr}(o)$ be the points-to set of address-taken
             variables $o$ in $IN_\ell$, where $\forall o \in Pts(y)$;
             $PG_{top} \hookleftarrow \{p \rightarrowtail (\gamma \wedge \varphi, o')\}, \forall(\gamma, o') \in \mathcal{P}_{adr}(o)$;
21     if $\ell, \varphi : (x_0, \ldots, x_n) = call\ f(v_1, \ldots, v_n)$ then
22         $PG_{top} \hookleftarrow \{x_i \rightarrowtail (\varphi, Trans(F, Pts(v_i)))\}, \forall i \in [1, n]$;
23     if $\ell, \varphi : fork(t, f)$ then
24         Continue;
```

4.2 Interference Dependence Analysis

At the stage of inter-thread reasoning, the goal is to identify all possible interference dependence as edges in the VFG, and compute the guards qualifying for the edges. Similar to the rules in Fig. 6, the core problem is to determine the pointer aliasing relationship of variables x and y for an edge of the interference dependence, $(q@\ell_1, p@\ell_2)$ between $*x = q$ and $p = *y$ across different threads. Different from data dependence, interference dependence is associated with the concurrent environment. Because the store at ℓ_1 should happen before the load at ℓ_2, we also need to enforce the order relation between the store and the load. To sum up, the guard Φ_{guard} for $(q@\ell_1, p@\ell_2)$ is defined as below:

$$\Phi_{guard}(q@\ell_1, p@\ell_2) = \Phi_{alias} \wedge \Phi_{ls}\,, \tag{1}$$

where Φ_{alias} indicates the conditions of the interference dependence through the same memory object, and Φ_{ls} ensures the order relations between stores and loads in a concurrent environment (as in Defn. 2(1)). We give more details in § 4.2.2.

4.2.1 Dependence Edges Computation. The high-level idea underlying our approach is to identify the variables that point to the shared memory objects, through which the thread interference happens. Below is an extended property of indirect value flows via interference dependence.

Property 1. *A variable q at one thread can flow to a variable p at another thread indirectly via interference dependence, only if q is stored to an escaped memory o first and p is loaded from the same memory object o afterwards.*

This property indicates that, a store statement $*x = q$ and a load statement $p = *y$ can share an interference dependence relation if and only if these statements reside in different threads, and x and y point to the same escaped object o. Thus, to identify the interference dependence, it suffices to check load statements and store statements in different threads that can access the same escaped objects. We develop an efficient algorithm that discovers the set of thread escaped objects and calculates the set of pointer variables pointing to the objects. Let *EspObj* be the set of escaped objects in the program, which are the objects in the VFG that can be reachable from the nodes in multiple threads. The pointer variables that point to an object o are called the pointed-to-by set of the object, denoted by $Pted(o)$. Given an escaped object $o \in EspObj$, we can compute $Pted(o)$ by collecting nodes that are reachable from o in the VFG generated by Alg. 1.

Example 4.2. In Fig. 2(a), we first capture the escaped object o_1 via the escape analysis. We can then compute the pointed-to-by sets $Pted(o_1)$ and $Pted(o_2)$ by traversing the VFG, which are $\{x, y\}$ and $\{b\}$ respectively. This information enables the adding of an edge of interference dependence, $(b@\ell_{13}, c@\ell_6)$, in the VFG by reasoning x is an alias of y.

A non-trivial question that immediately emerges is how to soundly compute these two sets, *EspObj* and *Pted*, to identify all the interference dependence edges. Note that, since the VFG is updated dynamically during the interference dependence analysis (i.e., new inter-thread value flows between stores and loads are introduced), the set of *EspObj* and their *Pted* information are also dynamically enlarged accordingly. In the example above, after the edge $(b@\ell_{13}, c@\ell_6)$ is updated in the VFG, the pointed-to-by set $Pted(o_2)$ can be updated from $\{b\}$ to $\{b, c\}$, considering the set of nodes that are reachable from o_2 in the VFG. Thus, $Pted(o_2)$ increases after the edge is added. Additionally, the memory object o_2 is escaped, because the o_2 becomes reachable from nodes in different threads and, thus, the size of $EspObj$ also increases. To sum up, the newly introduced value-flow edges of the interference dependence may enlarge the sets of both escaped objects and their pointed-to-by variables, which may, in turn, introduce new edges, forming *a cyclic dependence problem*, as also revealed in the previous work [52, 54].

Algorithm 2: Interference Dependence Analysis

```
   Input: A $VFG$ generated by Alg. 1
   Output: An interference-aware $VFG$
1  EscapeAnalysis($VFG$);
2  do
3      foreach store node $\ell_1, \varphi_1 : *x = q$ in $t \in \mathcal{T}$ do
4          foreach load node $\ell_2, \varphi_2 : p = *y$ in $t' \backslash t \in \mathcal{T}$ do
5              if $(x, \alpha), (y, \beta) \in Pted(o), o \in EspObj$ then
6                  Compute the dependence guard $\Phi_{guard}$;
7                  $\ell_1 : *x = q \xrightarrow{\Phi_{guard}} \ell_2 : p = *y$ in $VFG$;
8      Update $EspObj$ and their $Pted$ accordingly;
9      Update data dependence relations;
10 while No more interference edges are introduced;
11 return Interference-aware $VFG$;
12 Procedure EscapeAnalysis($VFG$):
13     Initialize $EspObj$ ;   // objects passed to the fork calls
14     foreach node $n$ in $VFG$ do
15         if $n$ is a store node $*x = q$ then
16             Let $o$ and $o'$ be the objects $x$ and $q$ point to;
17             if $o \in EspObj$ then
18                 $EspObj \hookleftarrow EspObj \cup \{o'\}$;
19     foreach escaped object $o$ in $EspObj$ do
20         Find the set of nodes $\mathcal{N}$ that reachable from $o$;
21         Let $\sigma$ the aggregated guards of edges during traversal;
           foreach node $n$ in $\mathcal{N}$ do
22             $Pted(o) \hookleftarrow Pted(o) \cup (n, \sigma)$;
23     return $EspObj$ and their $Pted$;
```

Thus, we develop an algorithm that updates the VFG by iteratively computing the escaped objects and their pointed-to-by set based on the updated edges, which terminates at a fixed point where no more value flows can be introduced. The set of escaped objects, *EspObj*, and their corresponding pointed-to-by set, *Pted*, thus are not finally updated. Compared to the previous escape analysis [11, 63], our analyses can capture the escaped objects due to the thread interactions between threads. Also, unlike in the previous work [60], our algorithm identifies the combined effects of thread interference by iteration, without using an exhaustive and expensive pointer analysis as a prior.

Alg. 2 describes the key steps. Lines 12-23 present the escape analysis. It first initializes the escaped objects, *EspObj*, by finding the objects passed to the fork calls, and then identifies the objects that are stored to an already escaped object via store statements (Lines 14-18). Next, the algorithm computes the pointed-to-by sets $Pted$ of $EspObj$ by finding nodes reachable from $EspObj$ and records the aggregated guards during the graph traversal (Lines 19-23). The guards represent the conditions of the pointed-to-by relations.

After the escape analysis, the algorithm identifies the interference dependence based on *EspObj* and *Pted* (Lines 3-9). Specifically, an edge $(q@\ell_1, p@\ell_2)$ is identified in the VFG, if x and y are both in the pointed-to-by set of an escaped object and these statements ℓ_1 and ℓ_2 are in different threads t and t' (Line 7). Next, we update *EspObj* and their *Pted* based on the updated edges (Line 8). The algorithm iteratively identifies new edges based on the dynamically updated *EspObj* and their *Pted*, which terminates until there are no more edges introduced. The analyses also update the intra-thread data dependence due to the side-effects of the identified interference dependence, in the measures similar to Alg. 1 (Line 9), so this stage is not a strict inter-thread analysis. However, the approach is thread-modular since it separates the intra-thread and inter-thread reasoning.

4.2.2 Dependence Guard Computation. We next give the details of how to compute the guard Φ_{guard} qualifying for an interference dependent edge in Alg. 2, following Defn. 2(1). The guard Φ_{guard} is the conjunction of the alias constraint Φ_{alias} and the load-store constraint Φ_{ls}.

Alias Constraint Φ_{alias}. Consider a value-flow edge $(q@\ell_1, p@\ell_2)$ between $\ell_1, \varphi_1 : *x = q$ and $\ell_2, \varphi_2 : p = *y$. The alias constraint Φ_{alias} comprises two parts: the conditions under which x and y point to the same memory object o, and the branch conditions of the statements ℓ_1 and ℓ_2. Equally, the conditions of x and y pointing to the same memory object o are the conditions under which the object o is pointed to by the variables x and y simultaneously. In Alg. 2, we know that the pointed-to-by conditions are α and β , which are the aggregated guards from the memory object o to the nodes x and y, computed while identifying the variables pointing to the escaped objects. To sum up, the alias constraint Φ_{alias} is $\varphi_1 \wedge \varphi_2 \wedge \alpha \wedge \beta$.

Example 4.3. In Fig. 2(a), the alias constraint Φ_{alias} for the edge $(b@\ell_{13}, c@\ell_6)$ first includes the conjunction of the branch conditions of these two statements $\ell_6 : c = *x$ and $\ell_{13} : *y = b$, which is $\theta_1 \wedge \neg\theta_1$. It should also include the conjuncted conditions of the pointer variables x and y in $Pted(o_1)$. By examining the aggregated guards in the VFG, we know that the condition is *true*. As a result, we conclude that $\Phi_{alias} = \theta_1 \wedge \neg\theta_1$.

Load-Store Constraint Φ_{ls}. According to the sequential consistency axioms in Defn. 2(1), if a stored variable $q@\ell_1$ flows to a loaded variable $p@\ell_2$, we need to enforce that (1) the execution order of ℓ_1 should be smaller than that of the load statement ℓ_2, and (2) there are no other concurrent store statements happening between them. Let s and l represent the store at ℓ_1 and the load at ℓ_2, and let $S(l)$ be the set of all stores that l is data dependent on. Formally, we use $O_l > O_s$ to denote the strict partial order between statements l and s, i.e., l is executed after s. Then, for an indirect value-flow edge from the store s to the load l, the Φ_{ls} is encoded basically as

$$\bigwedge_{s,s' \in S(l)} \left(O_s < O_l \bigwedge_{\forall s' \neq s} O_{s'} < O_s \vee O_l < O_{s'} \right) , \tag{2}$$

where the constraint $O_s < O_l$ ensures that the store s happens before the load l, and the constraint $O_{s'} < O_s \vee O_l < O_{s'}$ ensures that other concurrent stores s' can not happen between s and l. Note that, in practice, it is unnecessary to encode some order constraints between statements in the same thread, because we can quickly determine their order by traversing the control flow graph.

The load-store constraints Φ_{ls} between the successive nodes ℓ_{i-1} and ℓ_i over a value-flow path $\pi =< v_1@\ell_1, \ldots, v_n@\ell_n >$ result in the total order $<_\pi$. Note that since $<_\pi$ may violate the program order $<_P$, we will introduce additional constraints to enforce the program order $<_P$ (detailed in § 5.1).

Example 4.4. In Fig. 5(a), the Φ_{ls} for $(a@\ell_2, c@\ell_4)$ is $O_{\ell_2} < O_{\ell_4} \wedge O_{\ell_3} < O_{\ell_2}$. We do not need to encode $O_{\ell_3} > O_{\ell_4}$, since the statement ℓ_3 can only happen before ℓ_4 by the control flow information. Similarly, the Φ_{ls} for $(b@\ell_3, d@\ell_4)$ is $O_{\ell_2} < O_{\ell_3} \vee O_{\ell_4} < O_{\ell_2}$.

Summary. The thread-modular dependence analysis is efficient because it avoids explicitly enumerating exponential thread interleavings and generates the execution constraints without solving them eagerly. Instead, they are determined altogether at the bug checking stage. In addition, the algorithm iteratively identifies interference dependence by only reasoning the shared memory locations, avoids using an exhaustive and expensive pointer analysis as a prior, while remaining path sensitive, thereby significantly leading to a more effective algorithm than the previous [51, 54, 60].

5 Guarded Reachability Detection

Once the interference-aware value-flow graph is built, a number of concurrency bugs can be reduced to solving source-sink reachability problems over the guarded value-flow graph. For instance, to detect an inter-thread use-after-free bug, the source, *src*, is a *free* statement, the sink, *sink*, is a *use* statement, and the checking follows the value flows that connect *src* to *sink* across two or more threads. In this section, we address two problems to achieve precision and efficiency. The first is how to address the program order violation of a value-flow path, $<_P \not\Rightarrow <_\pi$, due to the intransitive interference dependence in Defn. 2(2). The second is how to efficiently solve the constraints that qualify for the value-flow paths.

5.1 Partially-ordered Value Flows Searching

A value-flow path connecting a source and a sink can be extracted by searching and transitively conjuncting the edges of both data dependence and interference dependence in the VFG. The searching of value flows is in similar ways to the previous work [10, 55, 61]. The intra-thread context-sensitivity is maintained using the clone-based function summary [39, 55]. Formally, a value-flow path π can be computed as follows:

$$\begin{aligned} &\pi = \langle v_1@\ell_1, v_2@\ell_2, \ldots, v_k@\ell_k\rangle, \\ &src = v_1@\ell_1, sink = v_k@\ell_k, \\ &(v_i@\ell_i, v_{i+1}@\ell_{i+1}) \text{ via either } dd \text{ or } id, 1 \le i < k\,, \end{aligned} \tag{3}$$

where the k represents the number of nodes in the value-flow path. We use dd and id to represent the data dependence and the interference dependence relationships, respectively.

Due to the intransitive property of interference dependence, the orders between the successive nodes over π may violate the program order, i.e., $<_P \not\Rightarrow <_\pi$. To solve this problem, the analysis has to take the correct program order into account, such as the control flows and the synchronization constructs (e.g., fork/join) semantics. Our main idea is to introduce the additional partial order constraints to enforce the program order $<_P$. The partial order constraints, denoted as Φ_{po}, together with the aggregated guards of value flows, are handed to an SMT solver to validate the realizability of the value-flow path π.

Partial Order Constraints Φ_{po}. The partial order consists of both the intra-thread partial order and the inter-thread partial order. The encoding schema is similar to the prior work [30, 31]. The intra-thread partial order follows the control flows of each thread. For instance, consider two successive nodes $v_1@\ell_1$ and $v_2@\ell_2$ in the same thread t. Their partial order constraint is written as $O_{\ell_1} < O_{\ell_2}$, if there is a valid control-flow path from ℓ_1 to ℓ_2. The inter-thread partial order follows the synchronization semantics between threads, which are related to the thread *fork* and *join* calls. A $fork$ call is the corresponding start operation of the newly forked thread, and a $join$ call is the exit operation of the joined thread. Therefore, the order of the nodes before a fork operation in the parent thread is smaller than all of the nodes in the corresponding child thread. Similarly, the order of the nodes after a join operation in the parent thread is larger than all of the nodes in the corresponding joined thread.

We encode these constraints Φ_{po} for each pair of successive nodes in a value-flow path π. For a pair of successive nodes, the analyses generate the corresponding partial order constraints if they share the program order relations above. Formally, we introduce a function $PO(v_i@\ell_i, v_{i+1}@\ell_{i+1})$ that returns the program order relations between statements ℓ_i and ℓ_{i+1} based on the rules above.

$$\Phi_{po}(\pi) = \bigwedge_{i\in[1,k-1],} \left(\bigwedge_{j\in[i,k-1]} PO(v_i@\ell_i, v_j@\ell_j) \right) . \tag{4}$$

Note that the partial order constraints Φ_{po} do not attempt to identify all the program orders enforced by other synchronization semantics like lock/unlock and wait/notify. However, the framework is generic enough to allow new synchronization semantics to be plugged in easily.

Example 5.1. In Fig. 5(b), Φ_{po} consists of program order $O_{\ell_1} < O_{\ell_2} \wedge O_{\ell_3} < O_{\ell_4}$. For a value-flow path $\langle a@\ell_2, b@\ell_3, b@\ell_4, a@\ell_1\rangle$, the Φ_{ls} is $O_{\ell_2} < O_{\ell_3} \wedge O_{\ell_3} < O_{\ell_4} \wedge O_{\ell_4} < O_{\ell_1}$. Since $O_{\ell_1} < O_{\ell_2}$ and $O_{\ell_2} < O_{\ell_1}$ conflict with each other, this irrealizable value-flow path can be pruned.

5.2 Constraints Solving

Finally, we encode all the constraints of a value-flow path into a formula Φ_{all}, which is the conjunction of the constraints Φ_{guards} qualifying for the def-use edges on the path, and the partial order constraints Φ_{po} enforcing Defn. 2(2).

$$\begin{aligned} &\Phi_{all}(\pi) = \Phi_{guards}(\pi) \wedge \Phi_{po}(\pi)\ , \text{ where} \\ &\Phi_{guards}(\pi) = \bigwedge_{i\in[1,k-1],} \Phi_{guard}(v_i@\ell_i, v_{i+1}@\ell_{i+1})\ . \end{aligned} \tag{5}$$

Next, the analysis feeds the collected constraints Φ_{all} to an SMT solver. Due to the complicated thread interference, we observe that the number of generated constraints can be extremely large, and some of those constraints can be complex to solve if without any optimization.

Our analysis optimizes the performance of constraint solving by three means. First, when resolving the data dependence and interference dependence (§ 4), we follow the previous work [8, 55] that uses lightweight semi-decision procedures to filter out conditions having any apparent contradictions. The optimization significantly reduces the computational and memory overhead for the subsequent analysis. Second, the constraints on different source-sink paths are independent of each other, which gives us the ability to leverage parallelization. This capability is important given the prevalence of multi-core processors. Third, for a complex query, our analysis can solve the constraints by the cube-and-conquer parallel SMT solving strategy [26].

Summary. There are two salient advantages to aggregate guards qualifying for value-flow paths and solve these constraints from different analysis domains together. First, a combined domain allows a more precise solution than one obtained by solving each domain separately [14]. Second, the better scalability is achieved by judiciously delaying the disjunctive reasoning of the realizability of the vulnerable paths until the phase of source-sink checking. Specifically, we avoid explicit case-splitting over the feasible thread interference with the exhaustive points-to information. Our approach is more scalable, since the SMT solver typically bypasses the analysis of all cases to prove a constraint satisfiable or unsatisfiable.

6 Implementation

We have implemented Canary on top of the LLVM compiler infrastructure and the Z3 SMT solver [15]. While the language in Fig. 3 has restricted language constructs, our implementations support the most features of C/C++, such as classes, dynamic memory allocation, references, and virtual functions. Arrays are considered monolithic. Currently, Canary supports POSIX thread in C and std::thread in C++.

Performance. When analyzing the interference dependence in Alg. 2, we have implemented a may-happen-in-parallel (MHP) analysis to prune the non-interference load and store statements for efficiency. Specifically, if a load statement and a store statement in two threads do not share MHP relations, by Defn. 1, it is impossible for them to share an interference dependence relation, thereby avoiding unnecessary reasoning. The MHP analysis itself has been studied for decades, and many highly scalable algorithms can be directly leveraged, which is orthogonal to our Alg. 2 but important to achieve high efficiency.

Practical programs often make asynchronous fork/join calls via function pointers. Thus, it is impossible to construct thread call graphs directly from the syntactic program description without analyzing function pointers. As shown in the previous work [25, 44], a precise call graph for C-like programs can be constructed only using the flow-insensitive analysis. Thus, we use Steensgaard's analysis for the thread call graph construction, which can be completed in almost linear time [59]. We also adopt the class hierarchy analysis to resolve the virtual function calls.

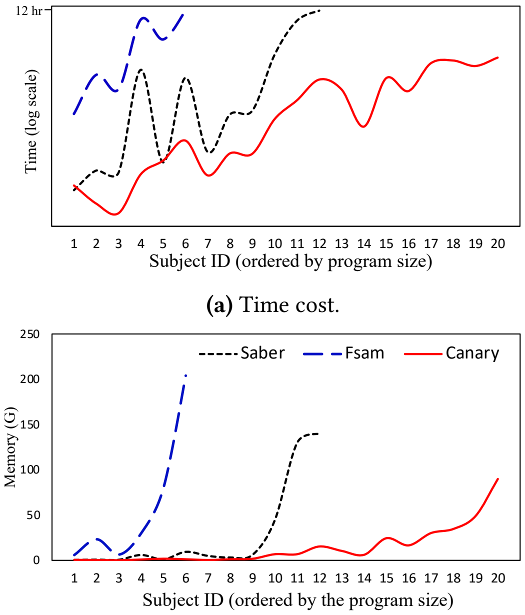
Soundiness. We follow the previous bug-finding techniques [55, 65] to engineer a soundy [41] bug detector, which means Canary is designed to find as many bugs as possible with low false positives, potentially at the cost of missing some bugs. Canary handles most language features in a sound manner, while it also applies some unsound choices as in the previous work [1, 55, 65]. For instance, we unroll each loop twice on the control flow graph, ignore inline assembly, and manually model parts of the standard C/C++ libraries. Besides, we assume the distinct parameters of a function do not alias with each other.

7 Evaluation

We evaluate the efficiency and the effectiveness of Canary in constructing the interference-aware value-flow graph and detecting concurrency bugs, by comparing it against two most related static tools, namely Saber [61] and Fsam [60].

The subjects we adopt include twenty real-life open-source C/C++ projects such as FireFox, MariaDB, and MySQL, which are commonly used in the concurrency analysis literature. Many of these projects are regularly and extensively scanned by the commercial tools as well as academic tools and, thus, are expected to have very high code quality. The sizes of these projects range from a few thousand LoC to close to nine million, with 936KLoC on average.

Canary is quite promising: it can finish a path-sensitive scan for about nine million lines of code in just 4.67 hours, with an average false positive rate around 26.67%. At the time of writing, it has found eighteen previously unknown concurrency bugs that have been confirmed by the developers with fourteen already fixed. This performance and precision are aligned with its design objectives and the common industrial requirement of detecting millions-of-LoC code in 5-10 hours with less than 30% false positives [5].



(a) Time cost.

(b) Memory cost.

Figure 7. Time and memory cost for building the VFG: Saber v.s. Fsam v.s. Canary.

All the experiments were performed on a computer with two 20 core Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz and 256GB physical memory running Ubuntu-16.04.

7.1 Comparison in Value-flow Graph Construction

First, we evaluate Canary's thread-modular data dependence analysis by comparing against two recent pointer-analysis-based techniques, Saber [61] and Fsam [60]. Saber performs an Andersen-style, flow-insensitive points-to analysis, which can trivially model the thread interference. Fsam is an Andersen-style, flow-sensitive pointer analysis for multi-threaded programs. They both perform an exhaustive points-to analysis to build the value-flow graph. To the best of our knowledge, these are the most precise and efficient tools for C/C++ multi-threaded programs we can get our hands on. We configure Saber and Fsam with their default settings. The timeout is set to twelve hours and the number of running threads is set to 1.

Fig. 7 shows the comparison of time and memory costs among Canary, Saber, and Fsam for constructing the value-flow graph. We can observe that Canary is much more scalable than Saber and Fsam. Canary can finish scanning all projects, but Saber times out on 9 projects and Fsam times

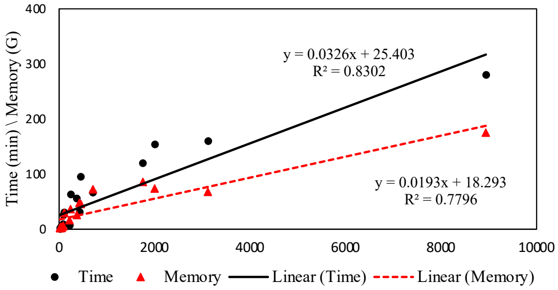


Figure 8. Scalability of Canary for bug hunting. The x-axis stands for the size of the project (KLoC) and the y-axis stands for the time cost (minutes) or the memory cost (GB).

out on 15 projects. To be specific, when the code size is larger than 50KLoC and 100KLoC, respectively, Fsam and Saber always run out of the time budget. In comparison, Canary can finish building the value-flow graph for 16 projects in less than one hour. On average, Canary is >15× and 180× faster than Saber and Fsam, respectively. At most, Canary is up to >70× and >500× faster than Saber and Fsam.

Besides, Canary requires significantly less memory compared to Saber and Fsam, as shown in Fig. 7b. For the subjects larger than 100 KLoC, Saber uses 130G additional memory compared to Canary. For the subjects larger than 50 KLoC, Fsam uses almost 200G additional memory, but it is still unable to finish building VFG for these subjects. Since our algorithm does not need to compute the exhaustive points-to results, Canary is much more efficient.

7.2 Comparison in Concurrency bug Detection

We evaluate the scalability and precision of Canary in the whole process of VFG construction and concurrency bug detection. The number of nested levels of calling context is set to six. For scalability, we do not list the runtime and memory consumptions of Saber and Fsam in the bug checking, because we have shown that they suffer from the scalability problem in the VFG construction. For precision, we compare the false positive rates among Canary, Fsam, and Saber. We planned to evaluate other bug-finding detection tools, such as Rader [12] and DCUAF [2]. However, they are either publicly unavailable or outdated for running in the environments we are able to set up. Among the inter-thread value-flow bugs, we choose to check inter-thread use-after-free, as it has become the most exploited memory errors, leading to many zero-day serious vulnerabilities [28, 67].

Scalability. To study the observed time and memory complexity of Canary, we adopt the curve fitting method. Fig. 8 reveals the fitting curves and their coefficients of determination R^2. The $R^2 \in [0, 1]$ depicts the statistical measure of how close the data are to the fitting curve. The closer R^2 is to 1, the better the fitting curve is. The figure suggests that Canary's time and memory cost grow almost linearly with R^2 around 0.8, and, thus, can scale up quite elegantly. Specifically, Canary can finish checking MySQL (3 MLoC) and FireFox (9 MLoC) in approximately 2.5 hours and 4.67 hours, respectively.

Table 1. Results of bug hunting.

Project	Size (KLoC)	Saber		Fsam		Canary	
		FP Rate	#Reports	FP Rate	#Reports	#FP	#Reports
1. lrzip	16	96.82%	63	93.75%	32	0	2
2. lwan	20	98.87%	89	100%	44	0	1
3. leveldb	21	100%	0	100%	0	1	1
4. darknet	29	100%	3,636	100%	144	0	0
5. coturn	39	100%	1,477	100%	368	0	2
6. httrack	49	100%	134	NA	NA	1	1
7. finedb	51	100%	421	NA	NA	0	1
8. tcpdump	85	100%	0	NA	NA	0	0
9. transmission	88	99.33%	299	NA	NA	0	2
10. celix	107	100%	3,782	NA	NA	0	0
11. redis	219	100%	0	NA	NA	0	0
12. git	239	NA	NA	NA	NA	0	0
13. zfs	367	NA	NA	NA	NA	0	1
14. HP-Socket	426	NA	NA	NA	NA	0	0
15. openssl	451	NA	NA	NA	NA	1	1
16. poco	705	NA	NA	NA	NA	0	0
17. mariadb	1,751	NA	NA	NA	NA	0	1
18. ffmpeg	2,003	NA	NA	NA	NA	0	0
19. mysql	3,118	NA	NA	NA	NA	0	0
20. firefox	8,938	NA	NA	NA	NA	1	2

NA means the analysis runs out of the time budget (12 hours).

Precision. Canary reports fifteen inter-thread use-after-free bugs with a false positive rate of 26.67%. All of the true positives are previously unknown and have been confirmed by our carefully checking manually. A stark contrast is that, Canary generates very few reports as shown in Tbl. 1, whereas Saber and Fsam report nearly 9,896 and 586 warnings, respectively. Since we are unable to manually inspect all of them, we randomly sample a hundred warnings for inspection if a project generates more than 100 warnings. Unfortunately, Saber and Fsam only found five and two true positives after the manual filtering, which were all found by Canary. However, because we can miss true bugs during manual checking the warnings found by Fsam and Saber due to subjectivity, we plan to make the implementation of inter-thread use-after-free checkers of Fsam and Saber available for the interested readers to examine. Concisely speaking, Canary is more precise because it builds more precise dependence relations and only takes the feasible interleaving executions into account.

7.3 Detected Real Concurrency Bugs

We have been using Canary to scan open-source projects extensively and continuously. At the time of writing, there were already eighteen concurrency bugs from dozens of open-source projects confirmed by developers, including the famous software systems such as firefox, leveldb, and transmission. Among the eighteen confirmed bugs, fourteen bugs have been fixed in the recent release versions of the software. A few vulnerabilities have been discovered on the key modules of the software and immediately fixed after being reported.

Canary found an old and latent inter-thread UAF vulnerability in transmission, missed by the developers, users, regression testing, and prior static analysis techniques. transmission is a popular BitTorrent client widely used by many Unix and Linux distributions including Ubuntu and Solaris. We retrospectively searched the commit history and found this bug has remained in concealment for almost eight years.

Canary can detect bugs of high complexity for which the original developers had to spend a considerable amount of time to confirm and even complained of the difficulty of fixing. For example, Canary detected an inter-thread use-after-free vulnerability in FireFox and LevelDB, respectively. The control-flow paths of these two bugs span across several functions and compiling units and the bugs may only be triggered in the rare thread schedules. The confirmation of the bugs was a team effort, through rounds of rejections before the final confirmation.

8 Related Work

Analysis of Concurrent Programs. The conventional data-flow analysis alternates between reasoning over intra-thread and inter-thread semantics, which is inherently wasteful since the analysis engine has to repetitively conduct reasoning over the similar intra-thread program regions [18]. Although many proposed algorithms account for the data dependence between the interleaved threads, the data-flow analysis itself still needs to propagate data-flow facts following intra-thread control flows [12, 17, 35, 60]. An early pointer analysis algorithm [54] for Clik programs only targets structured parbegin/parend parallelism, which solves data-flow problems and discovers the thread interactions by repeatedly reanalyzing each thread in each new analysis context until reaching a fixed point. However, non-lexically-scoped parallelism such as the fork/join model can significantly complicate the analysis. Rader [12] resorts to a data race detection engine to ensure the soundness of the sequential analysis by invalidating the data-flow facts influenced by concurrent writes. Fsam [60] follows the pre-computed thread-aware def-use chains to conduct pointer analysis. In comparison, Canary does not need an exhaustive Andersen-style pointer analysis as a prior and is also more precise than Fsam. To sum up, Canary checks bugs only following data and interference dependence in a value flow graph sparsely, thereby achieving more efficiency [55].

Model checking for concurrent programs has a long history. The key task is to identify redundant thread interleavings. A classical approach is partial order redunction [13, 20, 22]. Huang [27] proposes MCR (Maximal causality reduction) that captures the value of writes and reads in an execution trace to predict new traces. However, previous approaches target at the stateless model checking that could be considered as a form of systematic testing with a fixed input, as opposed to static analysis. Canary has a flavor of bounded model checking in the sense that it checks bugs as source-sink problems by exploring a value-flow graph under the finite unrolling of loops.

Symbolic execution, systematically exploring all feasible execution paths, also suffers seriously from state space explosion when being applied to multi-threaded programs [4, 23, 57]. Our approach allows the reduction of the redundant thread interleaving by checking value flows (a.k.a data dependence) without explicitly enumerating thread interleavings.

Concurrency Bug Detection. Concurrency bugs such as deadlock and race-like bugs are among the most notorious defects to deal with [42, 67]. Yang et al. [67] reveal that many concurrency vulnerabilities can be exploited to conduct severe attacks, such as privilege escalation and malicious code execution. This paper aims at detecting a category of concurrency bugs that could be expressed as source-sink properties. In the context of static approaches, much work on detecting data races has been done [6, 33, 40, 45, 62]. However, only 10% of true data races are harmful while most of them are benign [48]. Owl [69] relies on various data race detectors to disclose all possible racy program spots and further examine concurrency bugs. This work discovers 36 confirmed concurrency bugs among all 24,645 reported races in the Linux kernel. DCUAF [2] detects inter-thread UAF specifically for Linux drivers with a new method to find concurrent interface functions. Dynamic approaches typically include systematic testing [9, 10, 32, 66] or predictive analysis. For predictive analysis, they are based on approaches including the SMT-based methods [19, 27–29], partial-order-based methods [34, 43, 49, 53, 58], and distance-based exchangeable events [7] to predict additional executions from a single one. However, they can only predict other thread schedules under a specific input.

9 Conclusion

We have described Canary, our novel, principled approach to conducting interference-aware value-flow analysis for checking inter-thread value-flow bugs, which achieves both good precision and scalability for millions of lines of code. Canary is promising, having already pinpointed over eighteen previously-unknown concurrency bugs on a dozen of well-known open-source C/C++ projects.

Future work includes (1) enhancements to support more synchronization semantics like lock/unlock and signal/notify; (2) extension to relaxed memory models such as TSO/PSO; and (3) deployment of customized decision procedures to improve the efficiency of constraint solving.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their insightful comments. This work was partially funded by RGC16206517, ITS/440/18FP, Microsoft Donation, and Huawei Donation. Peisen Yao is the corresponding author.

References

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

[2] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 255–268. https://www.usenix.org/conference/atc19/presentation/bai

[3] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. 2007. Shape Analysis for Composite Data Structures. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 178–192. https://doi.org/10.1007/978-3-540-73368-3_22

[4] Tom Bergan, Dan Grossman, and Luis Ceze. 2014. Symbolic execution of multithreaded programs from arbitrary program contexts. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 491–506. https://doi.org/10.1145/2660193.2660200

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

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

[7] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. 2019. Detecting Concurrency Memory Corruption Vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 706–717. https://doi.org/10.1145/3338906.3338927

[8] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 363–374. https://doi.org/10.1145/1542476.1542517

[9] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2325–2342. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[10] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical Memory Leak Detection Using Guarded Value-Flow Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 480–491. https://doi.org/10.1145/1250734.1250789

[11] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/320384.320386

[12] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. 2008. Dataflow Analysis for Concurrent Programs Using Datarace Detection. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 316–326. https://doi.org/10.1145/1375581.1375620

[13] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. 1999. State Space Reduction Using Partial Order Techniques. *Int. J. Softw. Tools Technol. Transf.* 2, 3 (1999), 279–287. https://doi.org/10.1007/s100090050035

[14] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/567752.567778

[15] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[16] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 567–577. https://doi.org/10.1145/1993498.1993565

[17] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2013. Inductive data flow graphs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 129–142. https://doi.org/10.1145/2429069.2429086

[18] Azadeh Farzan and P. Madhusudan. 2007. Causal Dataflow Analysis for Concurrent Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4424)*, Orna Grumberg and Michael Huth (Eds.). Springer, 102–116. https://doi.org/10.1007/978-3-540-71209-1_10

[19] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. 2012. Predicting Null-Pointer Dereferences in Concurrent Programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 47, 11 pages. https://doi.org/10.1145/2393596.2393651

[20] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 110–121. https://doi.org/10.1145/1040305.1040315

[21] Malay Ganai, Dongyoon Lee, and Aarti Gupta. 2012. DTAM: Dynamic Taint Analysis of Multi-Threaded Programs for Relevancy. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 46, 11 pages. https://doi.org/10.1145/2393596.2393650

[22] Patrice Godefroid. 1997. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. In *Computer Aided Verification, 9th*

International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1254), Orna Grumberg (Ed.). Springer, 476–479. https://doi.org/10.1007/3-540-63166-6_52

[23] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. 2015. Assertion guided symbolic execution of multithreaded programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 854–865. https://doi.org/10.1145/2786805.2786841

[24] Ben Hardekopf and Calvin Lin. 2009. Semi-Sparse Flow-Sensitive Pointer Analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 226–238. https://doi.org/10.1145/1480881.1480911

[25] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer Society, 289–298. https://doi.org/10.1109/CGO.2011.5764696

[26] Marijn JH Heule, Oliver Kullmann, and Armin Biere. 2018. Cube-and-conquer for satisfiability. In *Handbook of Parallel Constraint Reasoning*. Springer, 31–59.

[27] Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 165–174. https://doi.org/10.1145/2737924.2737975

[28] Jeff Huang. 2018. UFO: Predictive Concurrency Use-after-Free Detection. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 609–619. https://doi.org/10.1145/3180155.3180225

[29] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

[30] Jeff Huang and Charles Zhang. 2012. LEAN: Simplifying Concurrency Bug Reproduction via Replay-Supported Execution Reduction. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 451–466. https://doi.org/10.1145/2384616.2384649

[31] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 141–152. https://doi.org/10.1145/2491956.2462167

[32] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP.2019.00017

[33] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. Static Data Race Detection for Concurrent Programs with Asynchronous Calls. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/1595696.1595701

[34] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 157–170. https://doi.org/10.1145/3062341.3062374

[35] Markus Kusano and Chao Wang. 2016. Flow-Sensitive Composition of Thread-Modular Abstract Interpretation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 799–809. https://doi.org/10.1145/2950290.2950291

[36] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28* 9 (September 1979), 690–691. https://www.microsoft.com/en-us/research/publication/make-multiprocessor-computer-correctly-executes-multiprocess-programs/

[37] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to Analysis with Efficient Strong Updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/1926385.1926389

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

[39] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and Scalable Context-Sensitive Pointer Analysis via Value Flow Graph. In *Proceedings of the 2013 International Symposium on Memory Management* (Seattle, Washington, USA) *(ISMM '13)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2464157.2466483

[40] Bozhen Liu and Jeff Huang. 2018. D4: fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 359–373. https://doi.org/10.1145/3192366.3192390

[41] Benjamin Livshits, Dimitrios Vardoulakis, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, José Amaral, Bor-Yuh Chang, Samuel Guyer, Uday Khedker, and Anders Møller. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58 (01 2015), 44–46. https://doi.org/10.1145/2644805

[42] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[43] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 145:1–145:29. https://doi.org/10.1145/3276515

[44] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2004. Precise Call Graphs for C Programs with Function Pointers. *Autom. Softw. Eng.* 11, 1 (2004), 7–26. https://doi.org/10.1023/B:AUSE.0000008666.56394.a1

[45] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 308–319. https://doi.org/10.1145/1133981.1134018

[46] Mangala Gowri Nanda and S. Ramesh. 2000. Slicing Concurrent Programs. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Portland, Oregon, USA) *(ISSTA '00)*. Association for Computing Machinery, New York, NY, USA, 180–190. https://doi.org/10.1145/347324.349121

[47] Mangala Gowri Nanda and S. Ramesh. 2006. Interprocedural Slicing of Multithreaded Programs with Applications to Java. *ACM Trans. Program. Lang. Syst.* 28, 6 (Nov. 2006), 1088–1144. https://doi.org/10.1145/1186632.1186636

[48] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/1250734.1250738

[49] Andreas Pavlogiannis. 2020. Fast, sound, and effectively complete dynamic race prediction. *Proc. ACM Program. Lang.* 4, POPL (2020), 17:1–17:29. https://doi.org/10.1145/3371085

[50] G. Ramalingam. 2000. Context-Sensitive Synchronization-Sensitive Analysis is Undecidable. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 416–430. https://doi.org/10.1145/349214.349241

[51] Venkatesh Prasad Ranganath and John Hatcliff. 2004. Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs. In *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2985)*, Evelyn Duesterwald (Ed.). Springer, 39–56. https://doi.org/10.1007/978-3-540-24723-4_4

[52] Martin C. Rinard. 2001. Analysis of Multithreaded Programs. In *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2126)*, Patrick Cousot (Ed.). Springer, 1–19. https://doi.org/10.1007/3-540-47764-0_1

[53] Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 747–762. https://doi.org/10.1145/3385412.3385993

[54] Radu Rugina and Martin Rinard. 1999. Pointer Analysis for Multithreaded Programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 77–90. https://doi.org/10.1145/301618.301645

[55] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 693–706. https://doi.org/10.1145/3192366.3192418

[56] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do I Use the Wrong Definition? DeFuse: Definition-Use Invariants for Detecting Concurrency and Sequential Bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 160–174. https://doi.org/10.1145/1869459.1869474

[57] Nishant Sinha and Chao Wang. 2010. Staged Concurrent Program Analysis. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 47–56. https://doi.org/10.1145/1882291.1882301

[58] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 387–400. https://doi.org/10.1145/2103656.2103702

[59] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

[60] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, Björn Franke, Youfeng Wu, and Fabrice Rastello (Eds.). ACM, 160–170. https://doi.org/10.1145/2854038.2854043

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

[62] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 205–214. https://doi.org/10.1145/1287624.1287654

[63] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 187–206. https://doi.org/10.1145/320384.320400

[64] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. 2012. Sound and precise analysis of parallel programs through schedule specialization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 205–216. https://doi.org/10.1145/2254064.2254090

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

[66] M. Xu, S. Kashyap, H. Zhao, and T. Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

[67] Junfeng Yang, Ang Cui, Salvatore J. Stolfo, and Simha Sethumadhavan. 2012. Concurrency Attacks. In *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*, Hans-Juergen Boehm and Luis Ceze (Eds.). USENIX Association. https://www.usenix.org/conference/hotpar12/workshop-program/presentation/yang

[68] 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 *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 218–229. https://doi.org/10.1145/1772954.1772985

[69] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. 2018. OWL: Understanding and Detecting Concurrency Attacks. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*. IEEE Computer Society, 219–230.