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

Yuandao Cai
The Hong Kong University of Science and Technology
Hong Kong, China
yciaibb@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.


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

ACM Reference Format:

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 analysis [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...
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.

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 >70x and >500x 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 (θ2){
  14.       free(b);
6.     int *c=*x;
  15.     }
7.     print(*c);  
  16. }
8. }
  9. }

(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 $o1$ 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 $o1$, since both are reachable from $o1$. This enables the discovery of the interference dependence edge (dashed line), because $b@ℓ_{13}$ may be stored to $o1$ and $c@ℓ_6$ may be loaded from $o1$ 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 $b@ℓ_{13}$ to $c@ℓ_6$ 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 $o1$, 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 $θ_1 \land \negθ_1$ include conjuncting the conditions for $x$ and $y$ pointing to an object $o1$ (true), and the branch conditions of statements $ℓ_6$ and $ℓ_{13}$ ($θ_1 \land \negθ_1$). Let $O_2 > O_1$ be the strict partial order between statements $ℓ_6$ and $ℓ_{13}$ that indicates $ℓ_6$ is executed after $ℓ_{13}$. 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 $b@ℓ_{13}$ to $c@ℓ_6$ without being overwritten by $a@ℓ_6$.

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 $O_{13} > O_6 \land O_{14} > O_{13}$, 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 nut-shell. Seemsly 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 if it were executing in isolation.

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 ∈ T represents a thread that corresponds to a context-sensitive fork site and comprises a finite number of functions F*. A label l ∈ 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 ∈ V and address-taken variables o ∈ 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 O are indirectly accessed via load and store statements, which take the top-level pointer variables, 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, 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 at most once shared access.

Value Flows. We say the value of a variable, a, flows to a variable, b, if a is assigned to b directly (via assignments, as b = a) or indirectly (via pointer dereferences, such as *y = a; x = 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@l, 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, (v1@l1, v2@l2). The guard D guard annotated on the edge indicates the condition under which the value flow happens. A path π = (v1@l1, . . . , vn@ln) 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 \( \pi = (v_1@f_1, \ldots, v_n@f_n) \) 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 \( v_i@f_{i-1} \) flows to \( v_i@f_i \) through the same memory location without being overwritten, where \( i \in [2, n] \).
2. For any \( f_i \) and \( f_j \), if \( f_i <_P f_j \), then \( f_i <_\pi f_j \), i.e., \( <_P \Rightarrow <_\pi \).

First of all, the imprecise dependence relations can lead to an irrealizable value-flow path \( \pi \), i.e., the stored variable \( v_{i-1}@f_{i-1} \) may not flow to the loaded variable \( v_i@f_i \) 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 \( a@f_2 \) can flow to \( c@f_2 \) only with the thread schedule, in which the statement \( f_3 \) happens after \( f_2 \) and the statement \( f_3 \) happens before \( f_2 \). 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 \( \pi \) 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 \( \pi \), while \( <_P \) represents the program order, including both the control flow and the synchronization semantics. In Fig. 5(b), a value-flow path \( (a@f_2, b@f_3, b@f_4, a@f_3) \) is not realizable, because the value-flow path leads to an invalid control-flow path from \( f_2 \) to \( f_4 \) where \( <_P \Rightarrow <_\pi \). 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 \( q@f_1 \) to \( p@f_2 \) between statements \( f_1 : *x = q \) and \( f_2 : p = *y \), 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 \( \Phi_{guard} \), 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 \( q@f_1 \) can flow to \( p@f_2 \) 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 \( \Phi_p \) 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 \( \Phi_{guard} \), following Defn. 2(1). Fig. 6 shows the basic rules for resolving the data dependence. The core problem is to calculate indirect flows from \( q@ \star x = q \) to \( p@\cdot p = *y \), which needs the points-to facts of the address-taken variables \( O \). The \( \Phi_{guard} \) 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 \( f_1 \) and \( f_2 \). 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
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 $V$ 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, $V$, and use the sets $IN_i$ and $OUT_i$ to hold the propagated pointer information for the address-taken variables, $O$, at the statement $t$ [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, $O$, 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.

![Figure 6. Basic rules for data dependence.](image)

**Algorithm 1: Data Dependence Analysis**

**Input:** A concurrent program $CP$  
**Output:** VFG with resolved intra-thread data dependence 

1. $CG \leftarrow$ construct the thread call graph;  
2. **foreach** function $F$ in reverse topological order of $CG$ **do**  
   **end**  
3. $T : \text{func}(v_1, \ldots, v_n) \Rightarrow (a_1, \ldots, a_m)^*$;  
4. $CFG \leftarrow$ Build intra-procedural control flow graph of $F$;  
5. **foreach** instruction $l$ in reverse post-order of $CFG$ **do**  
   **end**  
6. $\text{HandleEachInst}(l)$;  
7. Record side-effects on $v_l$ and construct $Trans$ for $F$;  
8. Resolve local data dependence relations in VFG;  
9. $\text{return}$ VFG;  

**Procedure** $\text{HandleEachInst}(l)$:

10. **if** $t, \varphi : p = \text{alloc}_o$ **then**  
11. $PG_{top} \leftarrow \{p \mapsto (\varphi, \text{alloc}_o)\}; \quad // \mapsto$: point to  
12. **end if**  
13. **if** $t, \varphi : p = q$ **then**  
14. $PG_{top} \leftarrow \{p \mapsto (\varphi \land \varphi, q)\}; \quad, \forall (\varphi, o) \in Pts(q)$;  
15. **end if**  
16. **if** $\varphi : *x = q$ **then**  
17. **if** $\text{Pts}(x)$ is a singleton then  
18. $IN_i \leftarrow (IN_i \setminus Pts(x))$;  
19. $OUT_i \leftarrow \{\text{Pts}(x) \mapsto (\varphi \land \varphi, o)\}; \quad, \forall (\varphi, o) \in Pts(q)$;  
20. Propagate $OUT_i$ to the successor(s);  
21. **end if**  
22. **if** $t, \varphi : p = *y$ **then**  
23. Let $P_{ad}(o)$ be the points-to set of address-taken variables $o$ in $IN_i$, where $\forall o \in Pts(y)$;  
24. $PG_{top} \leftarrow \{p \mapsto (\varphi \land \varphi, o')\}; \quad, \forall (\varphi, o') \in P_{ad}(o)$;  
25. **end if**  
26. **if** $t, \varphi : (x_0, \ldots, x_n) = \text{call } F(v_1, \ldots, v_n)$ **then**  
27. $PG_{top} \leftarrow \{x_i \mapsto (\varphi, Trans(F, Pts(v_i)))\}; \quad, \forall i \in [1, n]$;  
28. **end if**  
29. **if** $t, \varphi : \text{fork}(t, f)$ **then**  
30. **end if**  
31. 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, $(o_1, v_1, p_1 @ e)$ 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 $t_1$ should happen before the load at $e_2$, we also need to enforce the order relation between the store and the load. To sum up, the guard $\Phi_{guard}$ for $(o_1, v_1, p_1 @ e)$ is defined as below:

$$\Phi_{guard}(o_1, v_1, p_1 @ e) = \Phi_{alias} \land \Phi_{ls}. \quad (1)$$

where $\Phi_{alias}$ indicates the conditions of the interference dependence through the same memory object, and $\Phi_{ls}$ 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 \( \text{EspObj} \) 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 \( \text{Pted}(o) \). Given an escaped object \( o \in \text{EspObj} \), we can compute \( \text{Pted}(o) \) 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 \( \text{Pted}(o_1) \) and \( \text{Pted}(o_2) \) by traversing the VFG, which are \( \{x, y\} \) and \( \{b\} \) respectively. This information enables the adding of an edge of an interference dependence, \( (b@\ell_3, c@\ell_4) \), 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, \( \text{EspObj} \) and \( \text{Pted} \), 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 \( \text{EspObj} \) and their \( \text{Pted} \) information are also dynamically enlarged accordingly. In the example above, after the edge \( (b@\ell_3, c@\ell_4) \) is updated in the VFG, the pointed-to-by set \( \text{Pted}(o_2) \) can be updated from \( \{b\} \) to \( \{b, c\} \), considering the set of nodes that are reachable from \( o_2 \) in the VFG. Thus, \( \text{Pted}(o_2) \) 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 \( \text{EspObj} \) 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].

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, \( \text{EspObj} \), and their corresponding pointed-to-by set, \( \text{Pted} \), 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, \( \text{EspObj} \), 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 \( \text{Pted} \) of \( \text{EspObj} \) by finding nodes reachable from \( \text{EspObj} \) 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 $(q@t_1, p@t_2)$ is identified in the VFG, if $x$ and $y$ are both in the pointed-to-by set of an escaped object and these statements $t_1$ and $t_2$ 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 $\Phi_{guard}$ qualifying for an interference dependent edge in Alg. 2, following Defn. 2(1). The guard $\Phi_{guard}$ is the conjunction of the alias constraint $\Phi_{alias}$ and the load-store constraint $\Phi_{l}$.

**Alias Constraint $\Phi_{alias}$.** Consider a value-flow edge $(q@t_1, p@t_2)$ between $t_1, q_1 : x = q$ and $t_2, p_2 : y = p$. The alias constraint $\Phi_{alias}$ comprises two parts: the conditions under which $x$ and $y$ point to the same memory object $o$, and the branch conditions of the statements $t_1$ and $t_2$. 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 $\alpha$ and $\beta$, 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 $\Phi_{alias}$ is $\Phi_1 \land \Phi_2 \land \alpha \land \beta$.

**Example 4.3.** In Fig. 2(a), the alias constraint $\Phi_{alias}$ for the edge $(b@t_3, c@t_6)$ first includes the conjunction of the branch conditions of these two statements $t_6 : c = x \land x$ and $t_3 : y = b$, which is $\theta_1 \land \lnot \theta_1$. It should also include the conjunctioned 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 $\Phi_{alias} = \theta_1 \land \lnot \theta_1$.

**Load-Store Constraint $\Phi_{l}$.** According to the sequential consistency axioms in Defn. 2(1), if a stored variable $q@t_1$ flows to a loaded variable $p@t_2$, we need to enforce that (1) the execution order of $t_1$ should be smaller than that of the load statement $t_2$, and (2) there are no other concurrent store statements happening between them. Let $s$ and $l$ represent the store at $t_1$ and the load at $t_2$, and let $S(l)$ be the set of all stores that $l$ is data dependent on. Formally, we use $O_1 > O_s$ 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 $\Phi_{l}$ is encoded basically as

$$\bigcup_{s, l \in S(l)} \left( O_s < O_l \bigwedge \forall s' \exists s \left( O_s < O_s' \lor O_l < O_s' \right) \right).$$

where the constraint $O_s < O_l$ ensures that the store $s$ happens before the load $l$, and the constraint $O_s' < O_s \lor O_l < O_s'$ 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 $\Phi_{l}$ between the successive nodes $t_{i-1}$ and $t_i$ over a value-flow path $\pi = t_1@t_1, \ldots, t_n@t_n$ 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 $\Phi_{l}$ for $(a@t_2, c@t_4)$ is $O_{t_2} < O_{t_4} \land O_{t_4} < O_{t_4}$. We do not need to encode $O_{t_2} > O_{t_4}$, since the statement $t_4$ can only happen before $t_4$ by the control flow information. Similarly, the $\Phi_{l}$ for $(b@t_3, d@t_6)$ is $O_{t_3} < O_{t_6} \lor O_{t_6} < O_{t_6}$.

**Summary.** The thread-modular dependency 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, avoiding 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, $<_p \Rightarrow <_\pi$, 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 $\pi$ can be computed as follows:

$$\pi = (v_1@t_1, v_2@t_2, \ldots, v_k@t_k),$$

$$\text{src} = v_1@t_1, \text{sink} = v_k@t_k,$$

$$(v_i@t_i, v_{i+1}@t_{i+1}) \text{ via either } dd \text{ or } id, 1 \leq i < k,$$

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 $\pi$ may violate the program order, i.e., $\prec\pi\prec$. 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 $\prec_p$. The partial order constraints, denoted as $\Phi_{po}$, together with the aggregated guards of value flows, are handed to an SMT solver to validate the realizability of the value-flow path $\pi$.

**Partial Order Constraints $\Phi_{po}$.** 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 $v_1@t_1$ and $v_2@t_2$ in the same thread $t$. Their partial order constraint is written as $O_{t_1} \prec O_{t_2}$, if there is a valid control-flow path from $t_1$ to $t_2$. 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 $\Phi_{po}$ for each pair of successive nodes in a value-flow path $\pi$. 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 $PO(v_i@t_i, v_{i+1}@t_{i+1})$ that returns the program order relations between statements $t_i$ and $t_{i+1}$ based on the rules above.

$$\Phi_{po}(\pi) = \bigwedge_{i \in [1:k-1]} \left( \bigwedge_{j \in [j:k-1]} PO(v_i@t_i, v_j@t_j) \right).$$

Note that the partial order constraints $\Phi_{po}$ 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), $\Phi_{po}$ consists of program order $O_{t_1} < O_{t_2} \wedge O_{t_2} < O_{t_3}$. For a value-flow path $(a@t_2, b@t_3, b@t_4, a@t_5)$, the $\Phi_{po}$ is $O_{t_2} < O_{t_3} \wedge O_{t_3} < O_{t_4} \wedge O_{t_4} < O_{t_5}$. Since $O_{t_2} < O_{t_3}$ and $O_{t_3} < O_{t_4}$ conflict with each other, this irreducible value-flow path can be pruned.

### 5.2 Constraints Solving

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

$$\Phi_{all}(\pi) = \Phi_{guards}(\pi) \wedge \Phi_{po}(\pi),$$

$$\Phi_{guards}(\pi) = \bigwedge_{i \in [1:k-1]} \Phi_{guard}(v_i@t_i, v_{i+1}@t_{i+1}).$$

Next, the analysis feeds the collected constraints $\Phi_{all}$ 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.

**Soundness.** 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].

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...
We planned to evaluate other bug-finding detection tools, and we evaluate the scalability and precision of value-flow checking, that is to say, the better the fitting curve. The closer the fitting curves and their coefficients of determination are to 1, the better the data are to the fitting curve. The closer the fitting curves and their coefficients of determination are to 1, the better the data are to the fitting curve.

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.

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.

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 Tab. 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.

6.2 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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Size (KLoC)</th>
<th>CANARY</th>
<th>FSAM</th>
<th>Saber</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>Size (KLoC)</td>
<td>FP Rate #Reports</td>
<td>FP Rate #Reports</td>
<td>FP Rate #Reports</td>
</tr>
<tr>
<td>1. http</td>
<td>16</td>
<td>98.82% 63</td>
<td>93.75% 63</td>
<td>90.00% 63</td>
</tr>
<tr>
<td>2. hwan</td>
<td>20</td>
<td>98.87% 89</td>
<td>90.00% 89</td>
<td>90.00% 89</td>
</tr>
<tr>
<td>3. leveldb</td>
<td>21</td>
<td>100% 0</td>
<td>100% 0</td>
<td>100% 0</td>
</tr>
<tr>
<td>4. darknet</td>
<td>29</td>
<td>100% 0</td>
<td>100% 0</td>
<td>100% 0</td>
</tr>
<tr>
<td>5. coturn</td>
<td>30</td>
<td>100% 0</td>
<td>100% 0</td>
<td>100% 0</td>
</tr>
<tr>
<td>6. htttrack</td>
<td>49</td>
<td>100% 0</td>
<td>100% 0</td>
<td>100% 0</td>
</tr>
<tr>
<td>7. findlib</td>
<td>51</td>
<td>100% 0</td>
<td>100% 0</td>
<td>100% 0</td>
</tr>
<tr>
<td>8. txml</td>
<td>85</td>
<td>100% 0</td>
<td>100% 0</td>
<td>100% 0</td>
</tr>
<tr>
<td>9. transmission</td>
<td>88</td>
<td>99.33% 299</td>
<td>99.33% 299</td>
<td>99.33% 299</td>
</tr>
<tr>
<td>10. cellx</td>
<td>107</td>
<td>100% 0</td>
<td>100% 0</td>
<td>100% 0</td>
</tr>
<tr>
<td>11. redis</td>
<td>219</td>
<td>100% 0</td>
<td>100% 0</td>
<td>100% 0</td>
</tr>
<tr>
<td>12. git</td>
<td>239</td>
<td>NA  NA</td>
<td>NA  NA</td>
<td>NA  NA</td>
</tr>
<tr>
<td>13. zfs</td>
<td>367</td>
<td>NA  NA</td>
<td>NA  NA</td>
<td>NA  NA</td>
</tr>
<tr>
<td>14. HP-Socket</td>
<td>426</td>
<td>NA  NA</td>
<td>NA  NA</td>
<td>NA  NA</td>
</tr>
<tr>
<td>15. openssl</td>
<td>451</td>
<td>NA  NA</td>
<td>NA  NA</td>
<td>NA  NA</td>
</tr>
<tr>
<td>16. poco</td>
<td>705</td>
<td>NA  NA</td>
<td>NA  NA</td>
<td>NA  NA</td>
</tr>
<tr>
<td>17. mariadb</td>
<td>1,751</td>
<td>NA  NA</td>
<td>NA  NA</td>
<td>NA  NA</td>
</tr>
<tr>
<td>18. ffmpeg</td>
<td>2,003</td>
<td>NA  NA</td>
<td>NA  NA</td>
<td>NA  NA</td>
</tr>
<tr>
<td>19. mysql</td>
<td>3,118</td>
<td>NA  NA</td>
<td>NA  NA</td>
<td>NA  NA</td>
</tr>
<tr>
<td>20. firefox</td>
<td>8,938</td>
<td>NA  NA</td>
<td>NA  NA</td>
<td>NA  NA</td>
</tr>
</tbody>
</table>
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 reduction [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. DCUAL [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.


Canary: Practical Static Detection of Inter-thread Value-Flow Bugs
PLDI ’21, June 20–25, 2021, Virtual, Canada


