Conquering the Extensional Scalability Problem for Value-Flow Analysis Frameworks

Qingkai Shi  
The Hong Kong University of Science and Technology  
Hong Kong, China  
qshiaa@cse.ust.hk

Rongxin Wu  
Xiamen University  
Xiamen, China  
wurongxin@xmu.edu.cn

Gang Fan  
The Hong Kong University of Science and Technology  
Hong Kong, China  
gfan@cse.ust.hk

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

ABSTRACT
Modern static analyzers often need to simultaneously check a few dozen or even hundreds of value-flow properties, causing serious scalability issues when high precision is required. A major factor to this deficiency, as we observe, is that the core static analysis engine is oblivious of the mutual synergy among the properties being checked, thus inevitably losing many optimization opportunities. Our work is to leverage the inter-property awareness and to capture redundancies and inconsistencies when many properties are considered at the same time. We have evaluated our approach by checking twenty value-flow properties in standard benchmark programs and ten real-world software systems. The results demonstrate that our approach is more than 8× faster than existing ones but consumes only 1/7 of the memory. Such substantial improvement in analysis efficiency is not achieved by sacrificing the effectiveness: at the time of writing, thirty-nine bugs found by our approach have been fixed by developers and four of them have been assigned CVE IDs due to their security impact.

CCS CONCEPTS
• Software and its engineering → Software verification and validation.

KEYWORDS
Static bug finding, demand-driven analysis, compositional program analysis, value-flow analysis.

1 INTRODUCTION
Value-flow analysis [12, 30, 38, 41], which tracks how values are stored and loaded in a program, underpins the inspection of a very broad category of software properties, such as memory safety (e.g., null dereference, double free, etc.), resource usage (e.g., memory leak, file usage, etc.), and security properties (e.g., the use of tainted data). In addition, there are a large and growing number of domain-specific value-flow properties. For instance, mobile software requires that the personal information cannot be passed to an untrusted code [2], and, in web applications, tainted database queries are not allowed to be executed [43]. Fortify,1 a commercial static code analyzer, checks nearly ten thousand value-flow properties from hundreds of unique categories. Value-flow properties exhibit a very high degree of versatility, which poses great challenges to the effectiveness of general-purpose program analyzers.

Faced with such a massive number of properties and the need of extension, existing approaches, such as Fortify, CSA,2 and Infer,3 provide a customizable framework together with a set of property interfaces that enable the quick customization for new properties. For instance, CSA uses a symbolic-execution engine such that, at every statement, it invokes the callback functions registered for the properties. These callback functions are overwritten by the property-checker writers to collect the symbolic-execution results, such as the symbolic memory and the path conditions, so that we can judge the presence of any property violation at the statement. Despite the existence of many CSA-like frameworks, when high precision like path-sensitivity is required, existing static analyzers still cannot scale well with respect to a large number of properties to check, which we refer to as the extensional scalability issue. For example, our evaluation shows that CSA cannot path-sensitively check twenty properties for many programs in ten hours. Pinpoint [38], another recent analyzer, exhausted 256GB of memory for only eight properties.

We observe that a major factor for the extensional scalability issue is that, in the conventional extension mechanisms, such as that of CSA, the core static analysis engine is oblivious to the properties being checked. Although the property obliviousness gives the maximum flexibility and extensibility to the framework,
it also prevents the core engine from utilizing the property-specific analysis results for optimization. This scalability issue is slightly alleviated by a class of approaches that are property-aware and demand-driven [5, 25, 28]. These techniques are scalable with respect to a small number of properties because the core engine can skip certain program statements by understanding what statements are relevant or irrelevant to the properties. However, in these approaches, the semantics of properties are also opaque to each other. As a result, when the number of properties grows very large, the performance of the demand-driven approaches will quickly deteriorate because property-irrelevant program statements become fewer and fewer, such as in the case of Pinpoint. To the best of our knowledge, the number of literature specifically addressing the extensional scalability issue is very limited. Readers can refer to Section 7 for a detailed discussion.

In this work, we advocate an inter-property-aware design to relax the property-property and the property-engine opaqueness so that the core static analysis engine can exploit the mutual synergy among different properties for optimization. To check a value-flow property, instead of conforming to conventional callback interfaces, property-checker writers of our framework need to explicitly declare a simple property specification, which picks out source and sink values, respectively, as well as the predicate over these values for the satisfaction of the property. For instance, for a null dereference property, our property model only requires the checker writers to indicate where a null pointer may be created and where the null dereference may happen using pattern expressions, as well as a simple predicate that constrains the propagation of the null pointer. Surprisingly, given a set of properties specified in our property model, our static analyzer can automatically understand the overlaps and inconsistencies of the properties to check. Based on the understanding, before analyzing a program, we can make dedicated analysis plans so that, at runtime, the analyzer can share the analysis results on path-reachability and path-feasibility among different properties for optimization. The optimization allows us to significantly reduce redundant graph traversals and unnecessary invocations of the SMT solver, two critical performance bottlenecks of conventional approaches. We provide some examples in Section 2 to illustrate our approach.

We have implemented our approach, named Catapult, which is a new demand-driven and compositional static analyzer with the precision of path-sensitivity. Like a conventional compositional analysis [45], our implementation allows us to concurrently analyze functions that do not have calling relations. In Catapult, we have included all C/C++ value-flow properties that CSA checks by default. In the evaluation, we compared Catapult to three state-of-the-art bug-finding tools, Pinpoint, CSA, and Infer, using a standard benchmark and ten popular industrial-sized software systems. The experimental results demonstrate that Catapult is more than 8x faster than Pinpoint but consumes only 1/7 of the memory. It is as efficient as CSA and Infer in terms of both time and memory cost but is much more precise. Such promising scalability of Catapult is not achieved by sacrificing the capability of bug finding. In our experiments, although the benchmark software systems have been checked by numerous free and commercial tools, Catapult is still able to detect many previously-unknown bugs, in which thirty-nine have been fixed by the developers and four have been assigned CVE IDs. In summary, we make the following contributions:

• An inter-property-aware design for checking value-flow properties, which mitigates the extensional scalability issue.
• A series of cross-property optimization rules that can be made use of for general value-flow analysis frameworks.
• A detailed implementation and a systematic evaluation that demonstrates our high scalability, precision, and recall.

2 OVERVIEW

The key factor that allows us to conquer the extensional scalability problem is the exploitation of the mutual synergy among different properties. In this section, we first use two simple examples to illustrate this mutual synergy and then provide a running example used in the whole paper.

2.1 Mutual Synergy

We observe that the mutual synergy among different properties are primarily in the forms of path overlapping and path contradiction. In Figure 1a, to check the memory-leak bug, we need to track value flows from the newly-created heap pointer \( a \) to check if the pointer will be freed.\(^4\) To check the free-global-pointer bug, we track value flows from the global variable \( b \) to check if it will be freed.\(^5\) As illustrated in the figure, the value-flow paths to search for these two bugs overlap from the vertex \( c=\phi(a, b) \) to the vertex \( c=1 \). Being aware of the overlap, when traversing the graph from the vertex \( a=\text{malloc}() \) for the memory-leak bug, we record that the vertex \( c=\phi(a, b) \) cannot reach any "free" operation. Therefore, when checking the free-global-pointer bug, we can use this recorded information to immediately stop the graph traversal at the vertex \( c=\phi(a, b) \), thereby avoiding redundant graph traversals.

In Figure 1b, to check the memory-leak bug, we track value flows from the newly-created pointer \( a \) to where it is freed. To check the null-dereference bug, considering that the function \( \text{malloc} \) may return a null pointer when the memory allocation fails, we track the value flows from the same pointer \( a \) to where it is dereferenced. The two properties have an inconsistent constraint: the former requires \( a\neq 0 \) for \( a \) to be a valid heap pointer while the latter requires

\(^4\) In the paper, we say a pointer \( p \) is "freed" if it is used in the function call \( \text{free}(p) \). We will detail how to use the value-flow information to check bugs later.

\(^5\) Freeing a pointer pointing to non-heap memory (e.g., memory allocated by global variables) is buggy. See details in https://cwe.mitre.org/data/definitions/590.html.
where we check the null-dereference and the free-global-pointer bugs when checking the memory-leak bug.

when traversing the graph for checking the null-dereference bug, we check and record if the path condition $\gamma$ of the path from the vertex $a=\text{malloc}()$ to the vertex $b=a$ conflicts with the null pointer condition $a=0$. If the path condition $\gamma$ is satisfiable but conflicts with the null pointer condition $a=0$, i.e., the conjunction $\gamma \wedge a=0$ is unsatisfiable, we can conclude that the conjunction $\gamma \wedge a\neq 0$ must be satisfiable without an expensive constraint-solving procedure when checking the memory-leak bug.

2.2 A Running Example

Figure 3 shows a running example using the value-flow graph where we check the null-dereference and the free-global-pointer bugs following the workflow illustrated in Figure 2. Given a program, we first follow the previous work [12, 38, 42] to build the value-flow graph in order to check the two properties with the precision of path-sensitivity. Here, path-sensitivity means that when searching paths on the value-flow graph, we invoke an SMT solver to solve path conditions and other property-specific constraints to prune infeasible paths.

The Property Specifications. The users of our framework need to declaratively specify the value-flow properties, which consists of the simple descriptions of the sources, the sinks, and the predicates for triggering the bug. For instance, the specifications of the aforementioned two properties are described by the following two quadruples, respectively:

prop null-deref := (u = malloc(); a = malloc(); \_ = a; \_ = \_; u = 0; never)

prop free-glob-ptr := (glob; free(c); true; never)

Separated by the semicolons, the first and second components denote the descriptors of the source and the sink, respectively, specified using pattern expressions to represent the values used or defined in some program statements. The "don’t-care" values are written as underscores. In the running example, the source values of the properties null-deref and free-glob-ptr are the return pointer of the function malloc and the global pointer $g$, respectively. The sink value of the property null-deref is the dereferenced value $c$ at the statement "c=1". The sink values of the property free-glob-ptr are the freed values at the statements free($b$) and free($d$).

The third component is a property-specific constraint, representing the triggering condition of the bug. In our example, the constraint of the property null-deref is $v=0$, meaning that the value on a value-flow path should be a null pointer. The constraint of the property free-glob-ptr is true, meaning that the value on a value-flow path is unconstrained.

The built-in predicate "never" means that value-flow paths between the specified sources and sinks should never be feasible. Otherwise, a bug exists.

The Core Static Analysis Engine. Given these declarative specifications, our core engine automatically makes analysis plans before the analysis begins, including both the graph traversal plan and the optimization plan. In the example, we make the following optimization plans: (1) checking the property free-glob-ptr before the property null-deref; (2) when traversing the graph for the property free-glob-ptr, we record the vertices that cannot reach any sink vertex of the property null-deref. The graph traversal plan in the example is trivial, which is to perform a depth-first search on the value-flow graph from every source vertex of the two properties.

In Figure 3, when traversing the value-flow graph from the global pointer $g$ to check the property free-glob-ptr, the core engine visits all vertices except the vertex $p$ to look for "free" operations. According to the optimization plan, during the graph traversal, we record that the vertices $b$ and $d$ cannot reach any dereference operation.

To check the property null-deref, we traverse the value-flow graph from the vertex $p$. When visiting the vertex $b$ and the vertex $d$, since the previously-recorded information tells us that they cannot reach any sink vertices, we prune the subsequent paths from the two vertices.

It is noteworthy that if we check the property null-deref before the property free-glob-ptr, we only can prune one path from the vertex $c$ for the property free-glob-ptr based on the results of the property null-deref (see Section 4.2.1). We will further explain the rationale of our analysis plans in the following sections.

3 VALUE-FLOW PROPERTIES

This section provides a specification model for value-flow properties with the following two motivations. First, we observe that many property-specfic constraints play a significant role in performance optimization. The specific constraints of one property can be used to optimize checking of not just the property itself, but also of other properties being checked together.

Second, despite many studies on value-flow analysis [12, 30, 38, 41, 42], we still have a lack of general and extensible specification models that can maximize the opportunities of sharing analysis results across the processes of checking different properties. Some of the existing studies only focus on checking a specific property (e.g., memory leak [42]), while others adopt different specifications to check the same value-flow property (e.g., double free [12, 38]).
Table 1: Pattern expressions used in the specification.

| p | patterns | p | pattern list | v | values |
|---|---|---|---|---|---|
| $p_1, p_2, \ldots$ | $\text{patterns}$ | $v_0 = \text{sig}(v_1, v_2, \ldots)$ | $\text{pattern list}$ | $v_0 = v_1$ | $\text{values}$ |
| $\texttt{call}$ | $\text{load}$ | $\texttt{store}$ | $\texttt{assign}$ | $\texttt{globals}$ | $\text{symbol}$ |
| $\texttt{character string}$ | $\texttt{uninterested value}$ | $\texttt{socket leak}$ | $\texttt{closed network socket}$ | $\texttt{malloc}$ | $\texttt{free}$ |

Examples:

- $v = \text{malloc}(_{-})$: ret values of any statement calling malloc;
- $_{=} = \text{send}(_{-}, v, _{-})$: the 2nd arg of any statement calling send;
- $_{=} = *u$: dereferenced values at every load statement.

Preliminaries. In a similar style to existing approaches [29, 38, 42], we assume that the code of a program is in static single assignment (SSA) form, where every variable has only one definition [17]. Also, we say the value of a variable $a$ flows to a variable $b$ (or $b$ is data-dependent on $a$) if $a$ is assigned to $b$ directly (via assignments, such as $b=a$) or indirectly (via pointer dereferences, such as $p_0; p_1; b=*q$). Thus, a value-flow graph can be defined as a directed graph where the vertices are values in the program and the edges represent the value-flow relations. A path is called value-flow path if it is a path on the value-flow graph.

Property Specification. As defined below, we model a value-flow property as an aggregation of value-flow paths.

Definition 3.1 (Value-Flow Property). A value-flow property, $x$, is a quadruple: $\text{prop } x := \langle \text{scc}, \text{sink}, \text{psc}, \text{agg} \rangle$, where

- $\text{scc}$ and $\text{sink}$ are two pattern expressions (Table 1) that specify the sources and the sinks of the value-flow paths to track.
- $\text{psc}$ is a first-order logic formula, representing the property-specific constraint that every value on the value-flow path needs to satisfy.
- $\text{agg} \in \{\text{never}, \text{never-sim}, \text{must}, \ldots\}$ is an extensible predicate that determines how to aggregate value-flow paths to check the specified property.

In practice, we can use the quadruple to specify a wide range of value-flow properties. As discussed below, we put the properties into three categories, which are checked by aggregating a single, two, or more value-flow paths, respectively.

Single-Path Properties. We can check many program properties using a single value-flow path, such as the properties, null-deref and free-glob-ptr, defined in Section 2.2, as well as a broad range of taint issues that propagate a tainted object to a program point consuming the object [21].

Double-Path Properties. A wide range of bugs happen in a program execution because two program statements (e.g., two statements calling the function free) consecutively operate on the same value (e.g., a heap pointer). Typical examples include the use-after-free bug, a general form of the double-free bug, as well as the ones that operate on expired resources such as a closed file descriptor or a closed network socket. We check them using two value-flow paths from the same source value. As an example, the specification for checking the double-free bugs can be specified as

$$\text{prop double-free} := (v = \text{malloc}(_{-}); \text{free}(v); v \neq 0; \text{never-sim})$$

In the specification, the property-specific constraint $v \neq 0$ requires the initial value (or equivalently, all values) on the value-flow path is a valid heap pointer. This is because $v = 0$ means the function malloc fails to allocate memory and returns a null pointer. In this case, the “free” operation is harmless. The aggregate predicate “never-sim” means that two value-flow paths from the same pointer should never occur simultaneously. In other words, there is no control-flow path that goes through two different “free” operations on the same heap pointer. Otherwise, a double-free bug exists.

In Figure 3, for the two value-flow paths from the vertex $p$ to the two “free” operations, we can check the disjunction of their path conditions of the two paths, respectively.

All-Path Properties. Many bugs happen because we do not properly handle a value in all program paths. For instance, a memory-leak bug happens if there exists a feasible program path where we do not free a heap pointer. Other typical examples include many types of resource leaks such as the file descriptor leak and the socket leak. We check them by aggregating all value-flow paths from the same source value. As an example, we write the following specification for checking memory leaks:

$$\text{prop mem-leak} := (v = \text{malloc}(_{-}); \text{free}(v); v \neq 0; \text{must})$$

Compared to the property double-free, the only difference in the specification is the aggregate predicate. The aggregate predicate “must” means that the value-flow path from a heap pointer must be able to reach a “free” operation. Otherwise, a memory leak exists in the program.

In Figure 3, for the value-flow paths from the vertex $p$ to the two “free” operations, we can check the disjunction of their path conditions, i.e., $((\gamma_1 \land \gamma_2) \lor (\gamma_1 \land \gamma_3) \land (p \neq 0))$, to determine if a memory leak exists. Here, $(\gamma_1 \land \gamma_2)$ and $(\gamma_1 \land \gamma_3)$ are the path conditions of these two paths, respectively. The additional $\gamma_1$ is the condition on which the heap pointer is created.

4 INTER-PROPERTY-AWARE ANALYSIS

Given a number of value-flow properties specified as the quadruples $(\text{scc}; \text{sink}; \text{psc}; \text{agg})$, our inter-property-aware static analyzer searches the value-flow paths and checks bugs based on the path conditions, the property-specific constraint psc, and the predicate agg. In this paper, we concentrate on how to exploit the mutual synergy arising from the interactions of different properties to improve the searching efficiency of value-flow paths.

4.1 A Naive Static Analyzer

For multiple value-flow properties, a naïve static analyzer checks them independently in a demand-driven manner. As illustrated by Algorithm 1, for each value-flow property, the static analyzer traverses the value-flow graph from each of the source vertices. At each step of the graph traversal, we check if the property-specific
constraint psc is satisfiable with respect to the current path condition. If it is not satisfiable, we can stop the graph traversal along the current path. This path-pruning process is illustrated in the shaded part of Algorithm 1, which is a critical factor to improve the performance.

The key optimization opportunities come from the observation that the properties to check usually introduce overlaps and inconsistencies during the graph traversal, which cannot be exploited if they are independently checked as in the naïve approach.

4.2 Optimized Intra-procedural Analysis

As summarized in Table 2, given the property specifications, our inter-property-aware static analysis engine carries out two types of optimizations when traversing the value-flow graph: the first aiming at pruning paths and the second focusing on sharing paths when multiple properties are being checked. Each row of the table is a rule describing the specific precondition, the corresponding optimization, as well as its benefit. For the clarity of the discussion, we explain the rules in the context of processing a single-procedure program, followed by the discussion on the inter-procedural analysis in the next subsection.

4.2.1 Optimization Plan. Given the property specifications, we adopt Rules 1 – 4 in Table 2 to facilitate the path pruning.

Ordering the Properties (Rule 1). Given a set of properties with different source values, we need to determine the order in which they are checked. While we leave the finding of the perfect order that guarantees the optimal optimization to our future work, we observe that a random order can significantly affect the effectiveness of the path pruning and must be circumvented.

Let us consider the example in Figure 3 again. In Section 2.2, we have explained that if the property free-glob-ptr is checked before the property null-deref, we can prune the two paths from the vertex b and the vertex d when checking the latter. However, if we flip the checking order, only one path from the vertex c can be pruned. This is because, when checking the property null-deref, the core engine records that the vertex c cannot reach any sinks specified by the property free-glob-ptr.

Intuitively, what causes the fluctuation in the number of prunable paths is that the number of the “free” operations is more than the dereference operations in the value-flow graph. That is, the more sink vertices we have in the value-flow graph, the fewer paths we can prune for the property. Inspired by this intuition, the order of checking the properties is arranged according to the number of sink vertices. That is, the more sink vertices a property has in the value-flow graph, the earlier we check this property.

Recording Sink-Reachability (Rule 2). Given a set of properties \{prop_1, prop_2, \cdots\}, when checking the property prop_i by traversing the value-flow graph, we record if each visited vertex may reach a sink vertex of the property prop_j (j ≠ i). With the recorded information, when checking the property prop_j (j ≠ i) and visiting a vertex that cannot reach any of its sinks, we prune the paths from the vertex. Section 2.2 illustrates the method.

Recording the Checking Results of Property-Specific Constraints (Rules 3 & 4). Given a set of properties \{prop_1, prop_2, \cdots\}, when we check the property prop_i by traversing the value-flow graph, we record the path segments, i.e., a set of edges, that conflict with the property-specific constraint psc_i of the property prop_j (j ≠ i). When checking the property prop_j (j ≠ i), we prune the paths that include the path segments.

Let us consider the example in Figure 3 again. When traversing the graph from the vertex g to check the property free-glob-ptr, the core engine records that the condition of the edge from the vertex a to the vertex c, i.e., a ≠ 0, conflicts with the property-specific constraint of the property null-deref, i.e., a = 0. With this information, when checking the property null-deref, we can prune the subsequent path after the vertex c.

Thanks to the advances in the area of clause learning [6], we are able to efficiently compute some reusable facts when using SMT solvers to check path conditions and property-specific constraints. Specifically, we compute two reusable facts when a property-specific constraint psc_i conflicts with the current path condition pc.

When pc ∧ psc_i is unsatisfiable, we record the unsatisfiable core [22], which is a set of Boolean predicates in the path condition pc, e.g., \{y_1, y_2, \cdots\}, such that y_1 ∧ y_2 ∧ \cdots ∧ psc_i = false. Since the path condition pc is the conjunction of the edge constraint on the value-flow path, each predicate y_i corresponds to the condition of an edge e_i on the value-flow graph. Thus, we can record an edge set E = \{e_1, e_2, \cdots\}, which conflicts with the property-specific constraint psc_i. When checking the other property with the same property-specific constraint, if a value-flow path contains these recorded edges, we can prune the remaining paths.

In addition to the unsatisfiable cores, we also can record the interpolation constraints [14], which are even reusable for properties with a different property-specific constraint. In the above example, assume that the property-specific constraint psc_i is a = 0 and the predicate set \{y_1, y_2, \cdots\} is \{a + b > 3, b < 0\}. In the constraint solving phase, an SMT solver can refute the satisfiability of \((a + b > 3) ∧ (b < 0) ∧ (a = 0)\) by finding an interpolant \(γ′\) such that \((a + b > 3) ∧ (b < 0) ⇒ γ′\) but \(γ′ ⇒ ¬(a = 0)\). In the example, the interpolant \(γ′\) is \(a > 3\), which provides a detailed explanation why the set conflicts with the property-specific constraint a = 0. In addition, the interpolant also indicates that the \(γ\) set conflicts with many other constraints such as \(a < 0\) and \(a < 3\). Thus, given a property whose specific constraint conflicts with the interpolation constraint, it is sufficient to conclude that any value-flow path passing through the edge set E can be pruned.
Table 2: Rules of making analysis plans for a pair of properties.

| Optimization Plans | | Plan | Benefit |
|-------------------|-----------------|-----------------|-----------------|
| ID | Rule Name | Precondition | | |
| 1 | property ordering | #sink1 > #sink2 | check x before y | more chances to prune paths |
| 2 | result recording | check x before y | record vertices that cannot reach sink2 | prune paths at a vertex |
| 3 | | check x before y, psc1 = psc2 | record unsat cores that conflict with psc2 | prune paths if going through a set of edges |
| 4 | | check x before y, psc1 ≠ psc2 | record interpolants that conflict with psc2 | |

| Graph Traversal Plans | | Plan | Benefit |
|----------------------|-----------------|-----------------|-----------------|
| ID | Rule Name | Precondition | | |
| 5 | traversal merging | - | search from src1 for both properties | sharing path conditions |
| 6 | psc-check ordering | psc1 ∧ psc2 = psc1 | check psc1 first | if satisfiable, so is psc2 |
| 7 | | psc1 ∧ psc2 ≠ false | check psc1 ∧ psc2 | if satisfiable, both psc1 and psc2 can be satisfied |
| 8 | | psc1 ∧ psc2 = false | check any, e.g., psc1, first | if unsatisfiable, psc2 can be satisfied |

Ordering the Checks of Property-Specific Constraints (Rules 6 – 8). Since the graph traversals are merged for different properties, at a vertex, e.g., a in Figure 4, we have to check multiple property-specific constraints, e.g., a ≠ 0 for the property mem-leak and a = 0 for the property null-deref, with respect to the path condition. In a usual manner, we have to invoke an expensive SMT solver to check each property-specific constraint, significantly affecting the analysis performance when there are many properties to check. We mitigate this issue by utilizing various relations between the property-specific constraints, so that we can reuse SMT-solving results and reduce the invocations of the SMT solver.

Given two property-specific constraints, psc1 and psc2, we consider all three possible relations between them: psc1 ∧ psc2 = psc1, psc1 ∧ psc2 ≠ false, and psc1 ∧ psc2 = false. Since the property-specific constraints are often simple, these relations are easy to compute. These relations make it possible to check both psc1 and psc2 by invoking an SMT solver only once.

The first relation, psc1 ∧ psc2 = psc1, implies that any solution of the constraint psc1 also satisfies the constraint psc2. In this case, we first check if the constraint psc1 conflicts with the current path condition pc by solving the conjunction, pc ∧ psc1. If it is satisfiable, we can conclude that the conjunction, pc ∧ psc2, is also satisfiable.

The second relation, psc1 ∧ psc2 ≠ false, implies that there exists a solution that satisfying both the constraint psc1 and the constraint psc2. In this case, we first check the conjunction, pc ∧ psc1 ∧ psc2. If it is satisfiable, we can conclude that both of the constraints, psc1 and psc2, are satisfiable with respect to the path condition.

The third relation, psc1 ∧ psc2 = false, implies that there does not exist any solution that satisfies both the constraint psc1 and the constraint psc2. In this case, we check any of the constraints, psc1 and psc2, first. If the current path is feasible but the conjunction pc ∧ psc1 is not satisfiable, we can conclude that the conjunction pc ∧ psc2 can be satisfied without invoking SMT solvers.

4.2.2 Graph Traversal Plan. The graph traversal plan is to provide strategies of sharing paths among different properties.

Merging the Graph Traversal (Rule 5). We observe that many properties actually share the same or a part of source vertices and even the same sink vertices. If the core engine checks each property one by one, it will repetitively traverse the graph from the same source vertex for different properties. Therefore, our graph traversal plan merges the path searching processes for different properties.

As an example, in Figure 3, since the vertex p may represent either a heap pointer or a null pointer, checking both the property null-deref and the property mem-leak needs to traverse the graph from the vertex p. Figure 4 illustrates how the merged traversal is performed. That is, we maintain a property set during the graph traversal to record what properties the current path contributes to. Whenever visiting a vertex, we check if a property needs to be removed from the property set. For instance, at the vertex d, we may remove the property null-deref from the property set if we can determine the vertex d cannot reach any of its sinks. When the property set becomes empty, the graph traversal stops immediately.

Figure 4: Merging the graph traversal.
### 4.3 Modular Inter-procedural Analysis

Scalable program analyses need to exploit the modular structure of a program. They build function summaries, which are reused at different calling contexts [16, 45]. In Catapult, we can seamlessly extend our optimized intra-procedural analysis to modular inter-procedural analysis by exploring the local value-flow graph of each function and then stitching the local paths together to generate complete value-flow paths. In what follows, we explain our design of the function summaries.

In our analysis, for each function, we build three kinds of value-flow paths as the function summaries. They are defined below and, in a longer version of this paper [37], we formally prove the soundness of generating these function summaries. Intuitively, these summaries describe how function boundaries, i.e., formal parameters and return values, partition a complete value-flow path. Using the property **double-free** as an example, a complete value-flow path from the vertex \( p \) to the vertex \( \text{free}(b) \) in Figure 5 is partitioned to a sub-path from the vertex \( p \) to the vertex \( \text{ret } p \) by the boundary of the function \( \text{xmalloc} \). This sub-path is an output summary of the function \( \text{xmalloc} \) as defined below.

**Definition 4.1 (Transfer Summary).** A transfer summary of a function \( f \) is a value-flow path from one of its formal parameters to one of its return values.

**Definition 4.2 (Input Summary).** An input summary of a function \( f \) is a value-flow path from one of its formal parameters to a sink value in the function \( f \) or in the callees of the function \( f \).

**Definition 4.3 (Output Summary).** An output summary of a function \( f \) is a value-flow path from a source value to a return value of the function. The source value is in the function \( f \) or in the callees of the function \( f \).

After generating the function summaries, to avoid separately storing them for different properties, each function summary is labeled with a bit vector to record what properties it is built for. Assume that we need to check there properties, i.e., **null-deref**, **double-free**, and **mem-leak**, in Figure 5. We assign three bit vectors, \( 0b001 \), \( 0b010 \), and \( 0b100 \), to the three properties as their identities, respectively. As explained above, all three properties regard the vertex \( p \) as the source. The sink vertices for checking the properties **double-free** and **mem-leak** are the vertices \( \text{free}(b) \) and \( \text{free}(u) \). There are no sink vertices for the property **null-deref**. According to Definitions 4.1–4.3, we generate the following function summaries:

| Function | Summary Path | Label  | Type  |
|----------|--------------|--------|-------|
| \( \text{xmalloc} \) | \((p, \text{ret } p)\) | \(0b111\) | output |
| \( \text{xfree} \) | \((u, \text{ret } u)\) | \(0b111\) | transfer |
|          | \((u, \text{free}(u))\) | \(0b110\) | input |

The summary \((p, \text{ret } p)\) is labeled with \(0b111\) because all three properties regard \( p \) as the source. The summary \((u, \text{ret } u)\) is also labeled with \(0b111\) because the path does not contain any property-specific vertices and, thus, may be used to check all three properties. The summary \((u, \text{free}(u))\) is only labeled with \(0b110\) because we do not regard the vertex \( \text{free}(u) \) as a sink of the property **null-deref**.

When analyzing the main function, we concatenate its intra-procedural paths with summaries from its callees to generate a complete path. For example, a concatenation is illustrated below and its result is labeled by \(0b110\), meaning that the resulting path only works for the property **double-free** and the property **mem-leak**.

\[
(p, \text{ret } p)^{0b111} \circ (a) \circ (u, \text{free}(u))^{0b110} = (p, \text{ret } p, a, u, \text{free}(u))^{0b111\cup0b110} = (p, \text{ret } p, a, u, \text{free}(u))^{0b110}
\]

We observe that using value-flow paths as function summaries has a significant advantage for checking multiple properties. That is, since value flow is a common program relations, it can be reused across different properties. This is different from existing approaches that utilize state machine to model properties and generate state-specific function summaries [18, 25]. Since different properties usually have different states, compared to our value-flow-based function summaries, such state-specific function summaries have fewer opportunities to be reused across properties.

## 5 IMPLEMENTATION

In this section, we present the implementation details as well as the properties to check in our framework.

**Path-sensitivity.** We have implemented our approach as a prototype tool called Catapult on top of Pinpoint [38]. Given the source code of a program, we first compile it to LLVM bitcode,\(^6\) on which our analysis is performed. To achieve path-sensitivity, we build a path-sensitive value-flow graph and compute path conditions following the method of Pinpoint. The path conditions in our analysis are first-order logic formulae over bit vectors. A program variable is modeled as a bit vector, of which the length is the bit width (e.g., 32) of the variable’s type (e.g., int). The path conditions are solved by Z3 [19], a state-of-the-art SMT solver, to determine the path feasibility.

**Properties to check.** Catapult currently supports twenty C/C++ properties, briefly introduced in Table 3, defined by CSA.\(^7\) These

---

\(^6\) LLVM: [https://llvm.org/](https://llvm.org/).

\(^7\) More details of the properties can be found on [https://clang-analyzer.llvm.org/](https://clang-analyzer.llvm.org/).
properties include all CSA’s default C/C++ value-flow properties. All other default C/C++ properties in CSA but not in Catapult are simple ones that do not require a path-sensitive analysis. For example, the property security.insecureAPI.bcopy requires CSA report a warning whenever a program statement calling the function bcopy is found.

**Parallelization.** Our analysis is performed in a bottom-up manner, in which a function is always analyzed before its callers. After a function is analyzed, its function behavior is summarized as function summaries, which can be reused at different call sites. Thus, it is easy to run in parallel by analyzing functions without caller-callee relations independently [45]. Our special design for checking multiple properties together does not prevent the analysis from this parallelization strategy.

**Soundness.** We implement Catapult in a soundly manner [31]. This means that the implementation soundly handles most language features and, meanwhile, includes some well-known unsound design decisions as previous works [4, 12, 38, 42, 45]. For example, in our implementation, virtual functions are resolved by classic class hierarchy analysis [20]. However, we do not handle C style function pointers, inline assembly, and library functions. We also follow the common practice to assume distinct function parameters do not alias with each other [30] and unroll each cycle twice on the call graph and the control flow graph. These unsound choices significantly improve the scalability but have limited negative impacts on the bug-finding capability.

### 6 EVALUATION

To demonstrate the scalability of our approach, we compared the time and the memory cost of Catapult to three existing industrial-strength static analyzers. We also investigated the capability of finding real bugs in order to show that the increased scalability is not at the cost of sacrificing the bug-finding capability.

---

**Table 3: Properties to check in Catapult.**

| ID | Property Name                              | Brief Description                                      |
|----|--------------------------------------------|--------------------------------------------------------|
| 1  | core.CallAndMessage                        | Check for uninitialized arguments and null function pointers |
| 2  | core.DivideByZero                          | Check for division by zero                            |
| 3  | core.NonNullParamChecker                   | Check for null passed to function parameters marked with nonnull |
| 4  | core.NullDereference                       | Check for null pointer dereference                     |
| 5  | core.StackAddressEscape                    | Check that addresses of stack memory do not escape the function |
| 6  | core.UndefInBinaryOperatorResult           | Check for the undefined results of binary operations   |
| 7  | core.VLASize (Variable-Length Array)       | Check for declaration of VLA of undefined or zero size |
| 8  | core.uninitialized.ArraySubscript          | Check for uninitialized values used as array subscripts |
| 9  | core.uninitialized.Assign                   | Check for assigning uninitialized values                |
| 10 | core.uninitialized.Branch                  | Check for uninitialized values used as branch conditions |
| 11 | core.uninitialized.CapturedBlockVariable   | Check for blocks that capture uninitialized values      |
| 12 | core.uninitialized.UndefReturn             | Check for uninitialized values being returned to callers |
| 13 | cplusplus.NewDelete                        | Check for C++ use-after-free                           |
| 14 | cplusplus.NewDeleteLeaks                   | Check for C++ memory leaks                            |
| 15 | unix.Malloc                                | Check for C memory leaks, double-free, and use-after-free |
| 16 | unix.MismatchedDeallocator                 | Check for mismatched deallocators, e.g., new and free() |
| 17 | unix.utf8.String                           | Check for null pointers being passed to C string functions like strlen |
| 18 | alpha.core.CallAndMessageUnInitRefArg      | Check for uninitialized function arguments             |
| 19 | alpha.unix.Stream                          | Check for misuses of C stream APIs, e.g., an opened file is not closed |
| 20 | alpha.unix.Stream                          | Check stream handling functions, e.g., using a null file handle in fseek |

**Table 4: Subjects for evaluation.**

| ID | Program | Size (KLoC) | ID | Program | Size (KLoC) |
|----|---------|-------------|----|---------|-------------|
| 1  | mcf     | 2           | 13 | shadowsocks | 32          |
| 2  | bzip2   | 3           | 14 | webassembly | 75          |
| 3  | gzip    | 6           | 15 | transmission | 88          |
| 4  | parser  | 8           | 16 | redis     | 101         |
| 5  | vpr     | 11          | 17 | imagemagick | 358         |
| 6  | crafty  | 13          | 18 | python    | 434         |
| 7  | twolf   | 18          | 19 | glusterfs | 481         |
| 8  | eon     | 22          | 20 | icu       | 537         |
| 9  | gap     | 36          | 21 | openssl   | 791         |
| 10 | vortex  | 49          | 22 | mysql     | 2,030       |
| 11 | perlbench | 73         |    |          |             |
| 12 | gcc     | 135         |    |          |             |

**Baseline approaches.** We first compared Catapult to Pinpoint, a most recent value-flow analyzer with the precision of inter-procedural path-sensitivity [38]. In addition, we also compared Catapult to two widely-used open-source bug finding tools, CSA and Infer. All these tools in our evaluation were configured to use fifteen threads to take advantage of parallelization.

We also tried to compare Catapult to other static bug detection tools such as Saturn [45], Calysto [4], Semmle [3], Fortify, and Klocwork. However, they are either unavailable or not runnable on the experimental environment we are able to set up. The open-source static analyzer, FindBugs, was not included in our experiments because it only works for Java while we focus on the analysis of C/C++ programs. We did not compare Catapult to Tricoder [36], the static analysis platform from Google. This is because it uses CSA as the C/C++ analyzer, which is included in our experiments.

---

8Klocwork: [https://www.roguewave.com/products-services/klocwork/](https://www.roguewave.com/products-services/klocwork/).
9FindBugs Static Analyzer: [http://findbugs.sourceforge.net/](http://findbugs.sourceforge.net/).
Subjects for evaluation. To avoid possible biases on the benchmark programs, we included the standard and widely-used benchmarks, SPEC CINT200019 (ID = 1 ~ 12 in Table 4), in our evaluation. Meanwhile, to demonstrate the efficiency and effectiveness of Catapult on real-world projects, we also included ten industrial-sized open-source C/C++ projects (ID = 13 ~ 22 in Table 4), of which the size ranges from a few thousand to two million lines of code.

Environment. All experiments were performed on a server with eighty “Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz” processors and 256GB of memory running Ubuntu-16.04.

6.1 Comparing to Static Value-Flow Analyzer

We first compared Catapult to Pinpoint, the state-of-the-art value-flow analyzer. To quantify the effect of the graph traversal plan and the optimization plan separately, we also configured Catapult* to only contain the graph traversal plan.

In this experiment, we performed the whole program analysis by linking all compilation units of a project into a single file for the static analyzers to perform the cross-file analysis. Before the analysis, both Pinpoint and Catapult need to build the value-flow graph as the program intermediate representation. Since Catapult is built on top of Pinpoint, the pre-processing time and the size of value-flow graph are the same for both tools, which are almost linear to the size of a program [38]. Typically, for MySQL, a program with about two million lines of code, it takes twenty minutes to build a value-flow graph with seventy million nodes and ninety million edges.

Efficiency. The time and memory cost of checking each benchmark program is shown in Figure 6a. Owing to the inter-property-awareness, Catapult is about 8x faster than Pinpoint and takes only 1/7 of the memory on average. Typically, Catapult can finish checking MySQL in 5 hours, which is aligned with the industrial requirement of finishing an analysis in 5 to 10 hours [7, 32].

When the optimization plan is disabled, Catapult* is about 3.5x faster than Pinpoint and takes 1/5 of the memory on average. Compared to the result of Catapult, it implies that the graph traversal plan and the optimization plan contribute to 40% and 60% of the time cost reduction, respectively. Meanwhile, they contribute to 70% and 30% of the memory cost reduction, respectively. As a summary, the two plans contribute similar to the time cost reduction, and the graph traversal plan is more important for the memory cost reduction because it allows us to avoid duplicate data storage by sharing analysis results across different properties.

Using the largest subject, MySQL, as an example, Figure 6b illustrates the growth curves of both the time and the memory overhead when the properties in Table 3 are added into the core engine one by one. Figure 6b shows that, in terms of both time and memory overhead, Catapult grows much slower than Pinpoint and, thus, scales up quite gracefully.

It is noteworthy that, except for the feature of inter-property-awareness, Catapult follows the same method of Pinpoint to build value-flow graph and perform path-sensitive analysis. Thus, they have the similar performance to check a single property. Catapult performs better than Pinpoint only when multiple properties are checked together.

19SPEC CINT2000 benchmarks: https://www.spec.org/cpu2000/CINT2000/.

Effectiveness. Since both Catapult and Pinpoint check programs with the precision of inter-procedural path-sensitivity, as shown in the left part of Table 5, they produce a similar number of bug reports (# Rep) and false positives (# FP) for all the real-world programs except for the programs that Pinpoint fails to analyze due to the out-of-memory exception.

6.2 Comparing to Other Static Analyzers

To better understand the performance of Catapult in comparison to other types of property-unaware static analyzers, we also ran Catapult against two prominent and mature static analyzers, CSA (based on symbolic execution) and Infer (based on abductive inference). Note that Infer does not classify the properties to check as Table 3 but targets at a similar range of properties, such as null dereference, memory leak, and others.

In our experiment, CSA was run with two different configurations: one is its default configuration where a fast but imprecise range-based solver is employed to solve path conditions, and the other uses Z3 [19], a full-featured SMT solver, to solve path conditions. To ease the explanation, we denote CSA in the two configurations as CSA (Default) and CSA (Z3), respectively. Since CSA separately analyzes each source file and Infer only has limited capability of detecting cross-file bugs, for a fair comparison, all tools in the experiments were configured to check source files separately, and the time limit for analyzing each file is set to 60 minutes. Since a single source file is usually small, we did not encounter memory issues in the experiment but missed a lot of cross-file bugs as discussed later. Also, since we build value-flow graphs separately for each file and do not need to track cross-file value flows, the time cost of building value-flow graphs is almost negligible. Typically, for MySQL, it takes about five minutes to build value-flow graphs for all of its source code. This time cost is included in the results discussed below.

Note that we did not change other default configurations of CSA and Infer. This is because the default configuration is usually the best in practice. Modifying their default configuration may introduce more biases.

Efficiency (Catapult vs. CSA (Z3)). When both Catapult and CSA employ Z3 to solve path conditions, they have similar precision (i.e., full path-sensitivity) in theory. However, as illustrated in Figure 6c, Catapult is much faster than CSA and consumes a similar amount of memory for all of the subjects. For example, for MySQL, it takes about 36 hours for CSA to finish the analysis while Catapult takes only half an hour, consuming a similar amount of memory. On average, Catapult is 68x faster than CSA at the cost of only 2x more memory space. Both analyses can finish in 12GB of memory, available in common personal computers.

Efficiency (Catapult vs. CSA (Default) and Infer). As illustrated in Figure 6c, compared to both Infer and the default version of CSA, Catapult consumes a similar, sometimes a little higher, amount of time and memory. For instance, for MySQL, the largest subject program, all three tools finish the analysis in 40 minutes and consume about 10GB of memory. With similar efficiency, Catapult, as a fully path-sensitive analysis, is much more precise than the other two. The lower precision of CSA and Infer leads to many false positives as discussed below.
Effectiveness. In addition to the efficiency, we also investigate the bug-finding capability of the tools. The right part of Table 5 presents the results. Since we only perform file-level analysis in this experiment, the bugs reported by Catapult is much fewer than those in the left part of Table 5. Because of the prohibitive cost of manually inspecting all of the bug reports, we randomly sampled a hundred reports for the projects that have more than one hundred reports. Our observation shows that, on average, the false positive rate of Catapult is much lower than both CSA and Infer. In terms of recall, Catapult reports more true positives, which cover all those defects identified by CSA and Infer. For example, CSA often stops its analysis on a path after it finds the first bug. Together with the results on efficiency, we can conclude that Catapult is much more scalable than CSA and Infer because they have similar time and memory overhead but Catapult is much more precise and able to detect more bugs.

### 6.3 Detected Real Bugs

We note that the real-world software used in our evaluation is frequently scanned by commercial tools such as Coverity SAVET\(^\text{11}\) and, thus, is expected to have very high quality. Nevertheless, due to the high efficiency, precision, and recall, Catapult still can detect many deeply-hidden software bugs that existing static analyzers, such as Pinpoint, CSA, and Infer, cannot detect.

At the time of writing, thirty-nine previously-unknown bugs have been confirmed and fixed by the software developers, including seventeen null pointer dereferences, ten use-after-free or double-free bugs, eleven resource leaks, and one stack-address-escape bug. Four of them even have been assigned CVE IDs due to their significant security impact. We have made an online list for all bugs assigned CVE IDs or fixed by their original developers.\(^\text{12}\)

---

\(\text{11}\)Coverity Scan: https://scan.coverity.com/projects/.

\(\text{12}\)Detected real bugs: https://qingkaishi.github.io/catapult.html.
To the best of our knowledge, a very limited number of existing static analyses have studied how to statically check multiple program properties at once, despite that the problem is very important at an industrial setting. Goldberg et al. [26] make unsound assumptions and intentionally stop the analysis on a path after finding the first bug. Apparently, the approach will miss many bugs, which Pinpoint also cannot detect the bug because it is not memory-efficient and has to give up its analysis after the memory is exhausted.

7 RELATED WORK

As an example, Figure 7 presents a null-reference bug detected by Catapult in ImageMagick, which is a software suite for processing images. This bug is of high complexity, as it occurs in a function of more than 1,000 lines of code and the control flow involved in the bug spans across 56 functions over 9 files.

Since both CSA and Infer make many unsound trade-offs to achieve scalability, neither of them detects this bug. Pinpoint also cannot detect the bug because it is not memory-efficient and has to give up its analysis after the memory is exhausted.

Figure 7: A null-dereference bug in ImageMagick.

As an example, Figure 7 presents a null-reference bug detected by Catapult in ImageMagick, which is a software suite for processing images. This bug is of high complexity, as it occurs in a function of more than 1,000 lines of code and the control flow involved in the bug spans across 56 functions over 9 files.

Since both CSA and Infer make many unsound trade-offs to achieve scalability, neither of them detects this bug. Pinpoint also cannot detect the bug because it is not memory-efficient and has to give up its analysis after the memory is exhausted.

8 CONCLUSION

We have presented Catapult, a scalable approach to checking multiple value-flow properties together. The critical factor that makes our technique fast is to exploit the mutual synergy among the properties to check. Since the number of program properties to check is quickly increasing nowadays, we believe that it will be an important research direction to study how to scale up static program analysis for simultaneously checking multiple properties.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and Dr. Yaping Liu for their insightful comments. This work is partially funded by an MSRA grant, as well as Hong Kong GRF16230716, GRF16206517, ITS/215/16FP, and ITS/440/18FP grants. Rongxin Wu is partially supported by the NSFC Project No. 61902329 and is the corresponding author.

REFERENCES

[1] Sven Apel, Dirk Beyer, Vitaly Mordan, Vadim Mutilin, and Andreas Stahlbauer. 2016. On-the-fly decomposition of specifications in software model checking. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 349–361.
